JPL PUBLICATION 77-24, REVISION 1

Software Design and Documentation Language

Henry Kleine

(NASA-CR-162291) SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE, REVISION 1 (Jet Propulsion Lab.) 88 p HC A05/MF A01 CSCL 09B

N79-32875

Unclas
G3/61 35777

August 1, 1979

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 77-24, REVISION 1

Software Design and Documentation Language

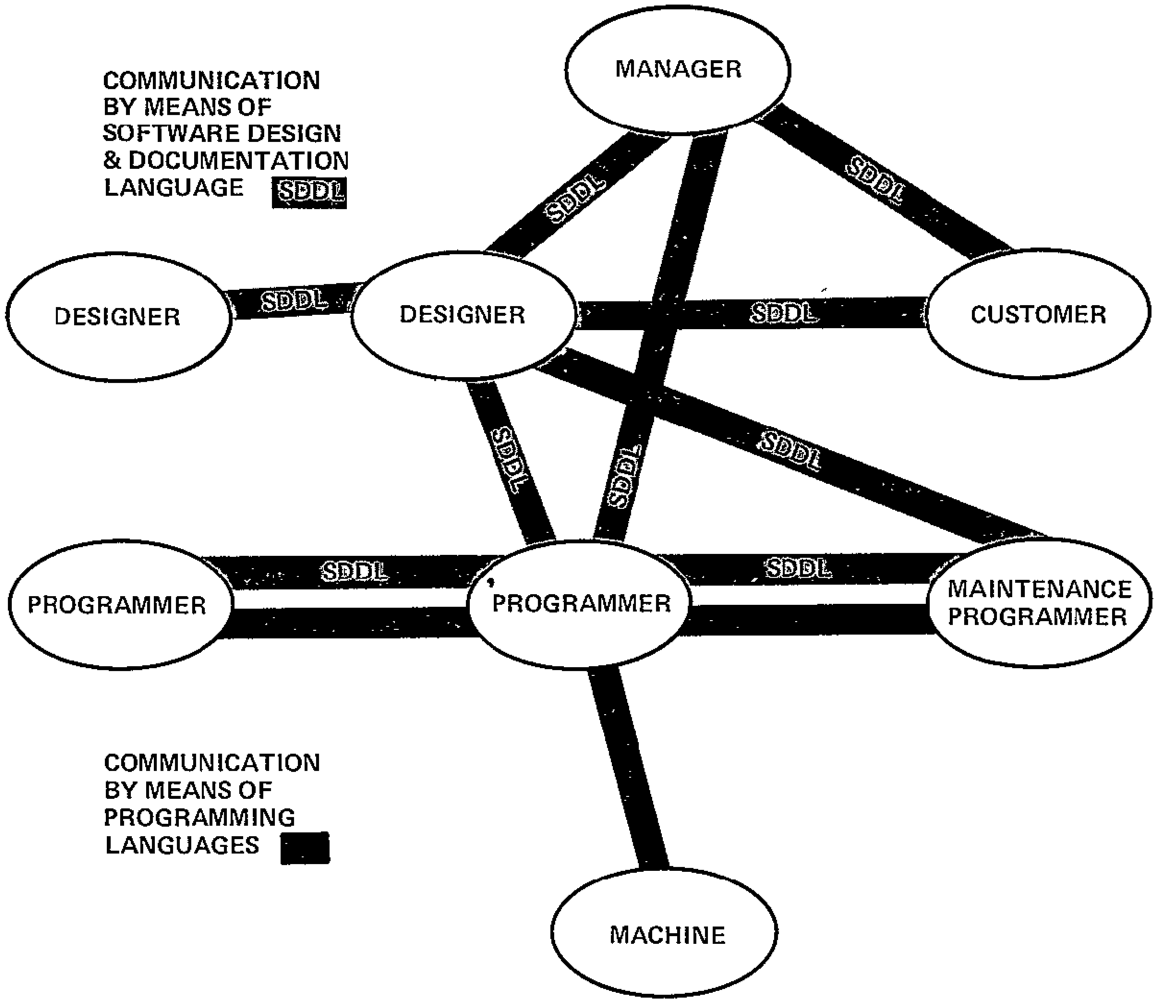
Henry Kleine

August 1, 1979

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract No NAS7-100


COMMUNICATION BY MEANS OF SOFTWARE DESIGN & DOCUMENTATION LANGUAGE SDDL
MANAGER
DESIGNER
DESIGNER
CUSTOMER
PROGRAMMER
PROGRAMMER
MAINTENANCE PROGRAMMER
MACHINE
SDDL
COMMUNICATION BY MEANS OF PROGRAMMING LANGUAGES

SOFTWARE DEVELOPMENT TEAM COMMUNICATIONS

PREFACE

The work described in this report was performed by the Information Systems Division (360) and the Systems Division (310) of the Jet Propulsion Laboratory.

ACKNOWLEDGMENT

Many aspects of the methodology for using SDDL, and enchancements to the language and the processor, evolved from its application to the design of two programs: the Vehicle Economy and Emissions Program (VEEP) and the Solar Array Manufacturing Industry Simulation (SAMIS). The current capabilities, present methodology, successful application, and future prospects of SDDL are, in large measure, due to the many contributions of the members of these design teams. For their many excellent suggestions, critical reviews of this document, critique of new processor capabilities, conscientious application of SDDL to the design tasks, and hours of philosophical discussion of the goals of a software design tool, I wish to express my thanks to Richard V. Morris, Donald A. Heimburger, Marcia A. Metcalfe, Bruce L. Kleine, Robert G. Chamberlain, Steve M. Jacobs, Robert L. Norton, and Gerhard J. Klose.

ABSTRACT

The objective of the Software Design and Documentation Language (SDDL) is to provide an effective communications medium to support the design and documentation of complex software applications. This objective is met by providing (1) a processor which can convert design specifications into an intelligible, informative machine-reproducible document, (2) a design and documentation language with forms and syntax that are simple, unrestrictive, and communicative, and (3) methodology for effective use of the language and processor.

The SDDL processor is written in the SIMSCRIPT II programming language and has been implemented on the UNIVAC 1108, the IBM 360/370, and Control Data machines.

CONTENTS

SECTION I

INTRODUCTION

The frontispiece is a conceptual view of the software development process. It identifies members of the software development team and shows the many communication links over which information must flow. The team members and the information flow shown in the diagram are a part of every software development project regardless of the number of individuals actually involved. Even when the entire task is done by a single person, it is still essential to have precise, accurate, orderly communication among the various roles the individual performs. With orderly communication, decisions made last month can be acted upon correctly this month, and valid information will be available later when maintenance responsibilities may have to be assumed by others.

The diagram also suggests that a computer programming language is a satisfactory communications medium for only a few links: primarily between programmer and machine, and secondarily among programmers. All other higher-level team communication requires less restrictive, more human-oriented media to be effective.

Historically, software development has suffered because of the lack of an effective communications medium for these high-level links. One may generalize that everyone has experienced some painful results of imprecise and/or incomplete communication in every aspect of life. Programmers suffer immediately when imprecise, incorrect, or incomplete directions are executed by the computer exactly as stated. Managers and customers are affected more seriously because bad communications at the design stage may compound the error by allowing the programming effort, with all its problems, to proceed toward an elusive or erroneous goal.

As long as the communication among members of the software development team remains fuzzy, the misunderstanding will continue and software development costs will be higher than they need be. Software maintenance gets into the act later, when maintenance programmers must deal with poorly written, out-of-date documentation, which, by Murphy's Law, is certain to be inconsistent where it matters.

Effective communication is not sufficient to insure efficient software development, but it is certainly necessary. Therefore, the Software Design and Documentation Language (SDDL) has been developed to satisfy this necessity.

A. SDDL OBJECTIVE

The objective of SDDL is to satisfy the communications requirements of the software design and documentation process. This objective is met by providing

(1) A processor which can translate design specifications, couched in SDDL syntax, into an intelligible, informative, machine-reproducible Software Design Document (SDD).

(2) A design and documentation language with forms and syntax that are simple, unrestrictive, and communicative.

(3) A methodology for effective use of the language and the processor.

B. SDDL PROCESSOR

The purpose of the SDDL processor is to translate the designer's creative thinking into an effective communications document. The processor must perform as many automatic functions as possible, thereby freeing the designer's energy for the creative design effort.

Some of the automatic functions which the processor, in its current state of development, performs are listed below.

1. Document Formatting

(1) Indentation by structure logic.

(2) Flow lines for accentuating structure escapes.

(3) Flow lines for accentuating module invocations.

(4) Line numbering and/or card sequencing for input deck editing.

(5) Logic error detection.

(6) Special handling for title pages and text segments.

(7) Input and output line continuation.

(8) Line splitting (i.e., printing part of the line so that the last character lines up at the right-hand margin).

2. Software Design Summary Information

(1) Table of contents showing all titles and modules, and the location of the summary tables provided by the processor.

(2) Module invocation hierarchy.

(3) Module cross reference (where each module is invoked).

(4) Cross reference tables for selected words or phrases appearing in the document. Selection is controlled by the user.

(5) Page reference numbers on module invocation statements.

3. Processor Control Capabilities

(1) Page width, length, numbering, heading, and ejection.

(2) Structure indentation amount.

(3) Deletion of preceding blank characters on input lines.

(4) Input line numbering sequence.

(5) Keyword specification.

(6) Selection of words for inclusion in the cross reference tables.

(7) Number of right-hand columns for card sequence numbers.

(8) Execution time options for suppressing selected processor features.

SECTION II

SDDL OVERVIEW

A. SDDL SYNTAX

The SDDL syntax consists of keywords (Table 2-1) used to invoke design structures, and a collection of directives (Table 2-2) which provide the user with control of processor actions such as indentation, page width, start of a new page, etc. Execution time options allow the user to selectively suppress design summary information.

Table 2-1. Default SDDL Structure Keywords

	INITIATOR	TERMINATOR	ESCAPE	SUBSTRUCTURE
MODULE	PROGRAM	ENDPROGRAM	EXITPROGRAM	
	PROCEDURE	ENDPROCEDURE	EXITPROCEDURE	
BLOCK	IF	ENDIF		ELSE ELSEIF
	SELECT	ENDSELECT		CASE
	LOOP	ENDLOOP REPEAT	EXITLOOP CYCLE	

MODULE INVOCATION KEYWORDS	CALL, DO

Table 2-2. SDDL Directive Keywords

#DEFINE	#EJECT
#TERMINATE	#SAMEPAGE
#MARK	#HEADING
#STRING	#PAGENUMBER
#TITLE	#PAGELENGTH
#TEXT	#LINENUMBER
#END	#WIDTH
#INDENT	#SEQUENCE
#BLANKS	

Input to the SDDL processor consists of a sequence of SDDL statements. An SDDL statement begins and ends with a line (or record) of the input medium. Continuation may be explicitly indicated by an ampersand (&) as the last non-blank character of the line. Continued lines are concatenated into a single statement for processing. Any natural language text, including a blank line, is an acceptable SDDL statement. Keywords are recognized only in context, i.e., only when they appear as the first word of the input statement.

The user is provided complete control of the choice of keywords by an SDDL directive which allows unlimited addition or deletion of keywords. User control of keyword selection is one of the most important features of SDDL because it allows the designer to command the capabilities of the processor in the way which is best suited to communicating the intent of the document.

A complete description of the SDDL semantics is given in Section IV.

B. SDDL STRUCTURES

The basic forms of the language are the module and block structures and the Module Invocation statement. A design is stated in terms of modules that represent problem abstractions which are complete and independent enough (relative to the level of the design) to be treated as separate problem entities. Modules are the highest-level structure. They may not be nested. Descriptive names are given to the modules, and their interrelationships are stated explicitly by the Module Invocation statements. A Module Invocation statement is the equivalent of the subroutine CALL statement in a programming language.

Blocks are the lower-level structures. They are used to build representations of abstractions which should (relative to the specific design) be a part of and appear in the higher-level abstraction represented by the module. Thus blocks must be nested within modules and may be nested within other blocks to any reasonable (i.e., understandable) depth. Examples of the use of blocks are the representations of Structured Programming concepts such as IF-THEN-ELSE and LOOP-REPEAT.

Both kinds of structures may have up to four parts:

(1)	Initiator	(required)
(2)	Terminator	(optional)
(3)	Escape	(optional)
(4)	Substructure	(optional)

Structure parts are specified by statements which begin with a keyword that has been defined as the part name. Table 2-1 displays the SDDL default keywords for both kinds of structures and their corresponding structure parts.

The actions taken by the processor in response to keyword statements are fully explained in Section IV and summarized in Figure 2-1. These actions are quite simple but very effective for communicating design information. Indentation of statements within structures and flow lines that highlight structure escapes and module invocations provide visual, two-dimensional information display which captures all of the advantages offered by flowcharts without their attendant disadvantages and constraints.

A simple illustration is presented in the example below.

In most of the following examples, the SDDL input statements are shown with the resulting output produced by the processor. In practice, the input source listing is rarely needed. Where the source statements are shown, as in the example below, it should be understood that the line numbering is not part of the input statement.

Example: Structured programming constructs

As input:

```
 1       PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
 2
 3       (THE LINE ABOVE IS A MODULE INITIATOR STATEMENT WHICH ESTABLISHES
 4       "EXAMPLE" AS THE NAME OF THIS PROGRAM/MODULE)
 5
 6          NOTE:  THE PARENTHESES IN THIS EXAMPLE ARE USED FOR
 7       COMMENTARY PURPOSES ONLY AND HAVE NO EFFECT ON THE SDDL
 8       PROCESSOR OR ITS OPERATION.
 9
10       IF THIS CONDITION IS TRUE  (BLOCK INITIATOR "IF")
11       ACT ON THIS STATEMENT  (PASSIVE STATEMENT)
12
13       ELSE  (SUBSTRUCTURE STATEMENT FOR "IF")
14       ACT ON THE FOLLOWING STATEMENTS  (ANOTHER PASSIVE STATEMENT)
15
16       LOOP FOR INDEX = 1 TO SOMETHING  (BLOCK INITIATOR "LOOP")
17          (PASSIVE STATEMENTS CAN BE PLACED ANYWHERE)
18       CALL SUBROUTINE  (MODULE INVOCATION STATEMENT)
19       THE NAME OF THE MODULE INVOKED IN THE PREVIOUS STATEMENT
20       IS "SUBROUTINE"  (PASSIVE STATEMENT)
21       IF THERE IS NOTHING LEFT TO DO  (NESTED BLOCK INITIATOR "IF")
22       EXITLOOP  (ESCAPE STATEMENT "LOOP")
23       ENDIF  (TERMINATOR STATEMENT NESTED "IF")
24       ENDLOOP  (TERMINATOR STATEMENT "LOOP")
25
26       ENDIF  (TERMINATOR STATEMENT "IF")
27
28       ENDPROGRAM  (MODULE TERMINATOR STATEMENT "PROGRAM")
29
30       PROCEDURE SUBROUTINE
31
32          NOTE:  A MODULE INITIATOR STATEMENT CAUSES THE START OF A NEW PAGE.
33       ALSO NOTE THAT "PROCEDURE" CAN BE USED AS A SYNONYM FOR "PROGRAM".
34
35       SELECT CASE BASED ON SOME CRITERION  (BLOCK INITIATOR "SELECT")
36
37       CASE 1: CHECK FOR SUBROUTINE ABORT (SUBSTRUCTURE STATEMENT FOR "SELECT")
38       IF THERE IS NO MORE DATA TO BE READ  (BLOCK INITIATOR "IF")
39       EXITPROCEDURE  (ESCAPE STATEMENT "PROCEDURE")
40       ENDIF
41
42       CASE 2: CHECK FOR SUBROUTINE ERROR (SUBSTRUCTURE STATEMENT FOR "SELECT")
43       IF AN ERROR OCCURS  (BLOCK INITIATOR "IF")
44       PRINT AN ERROR MESSAGE  (PASSIVE STATEMENT)
45       ENDIF
```

```
 46
 47      CASE 3: INVOKE ANOTHER SUBROUTINE  (SUBSTRUCTURE STATEMENT FOR "SELECT")
 48      DO ANOTHER SUBROUTINE  (MODULE INVOCATION STATEMENT)
 49      NOTE:  "DO" IS A SYNONYM FOR "CALL"  (PASSIVE STATEMENT)
 50
 51      ENDSELECT  (TERMINATOR STATEMENT "SELECT")
 52
 53      ENDPROCEDURE (MODULE TERMINATOR STATEMENT "PROCEDURE")
```

As output:

```
                              TABLE OF CONTENTS                           PAGE    I
 PAGE   LINE +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   1      1    PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
   2     30    PROCEDURE SUBROUTINE

   3         MODULE REFERENCE TREE

   4         MODULE - CROSS REFERENCE LISTING
```

```
LINE                                                                       PAGE    1
   1 PROGRAM EXAMPLE TO DEMONSTRATE THE BASIC SDDL STRUCTURES
   2
   3    (THE LINE ABOVE IS A MODULE INITIATOR STATEMENT WHICH ESTABLISHES
   4    "EXAMPLE" AS THE NAME OF THIS PROGRAM/MODULE)
   5
   6       NOTE:  THE PARENTHESES IN THIS EXAMPLE ARE USED FOR
   7    COMMENTARY PURPOSES ONLY AND HAVE NO EFFECT ON THE SDDL
   8    PROCESSOR OR ITS OPERATION.
   9
  10    IF THIS CONDITION IS TRUE  (BLOCK INITIATOR "IF")
  11       ACT ON THIS STATEMENT  (PASSIVE STATEMENT)
  12
  13    ELSE  (SUBSTRUCTURE STATEMENT FOR "IF")
  14       ACT ON THE FOLLOWING STATEMENTS  (ANOTHER PASSIVE STATEMENT)
  15
  16       LOOP FOR INDEX = 1 TO SOMETHING  (BLOCK INITIATOR "LOOP")
  17             (PASSIVE STATEMENTS CAN BE PLACED ANYWHERE)
  18          CALL SUBROUTINE  (MODULE INVOCATION STATEMENT)-------------->(  2)
  19          THE NAME OF THE MODULE INVOKED IN THE PREVIOUS STATEMENT
  20          IS "SUBROUTINE"  (PASSIVE STATEMENT)
  21          IF THERE IS NOTHING LEFT TO DO  (NESTED BLOCK INITIATOR "IF")
  22       <-----EXITLOOP  (ESCAPE STATEMENT "LOOP")
  23          ENDIF  (TERMINATOR STATEMENT NESTED "IF")
  24       ENDLOOP  (TERMINATOR STATEMENT "LOOP")
  25
  26    ENDIF  (TERMINATOR STATEMENT "IF")
  27
  28 ENDPROGRAM  (MODULE TERMINATOR STATEMENT "PROGRAM")
```

```
LINE                                                                        PAGE    2
  30 PROCEDURE SUBROUTINE
  31
  32        NOTE:  A MODULE INITIATOR STATEMENT CAUSES THE START OF A NEW PAGE.
  33     ALSO NOTE THAT "PROCEDURE" CAN BE USED AS A SYNONYM FOR "PROGRAM".
  34
  35     SELECT CASE BASED ON SOME CRITERION  (BLOCK INITIATOR "SELECT")
  36
  37     CASE 1: CHECK FOR SUBROUTINE ABORT (SUBSTRUCTURE STATEMENT FOR "SELECT")
  38        IF THERE IS NO MORE DATA TO BE READ  (BLOCK INITIATOR "IF")
  39 <---------EXITPROCEDURE  (ESCAPE STATEMENT "PROCEDURE")
  40        ENDIF
  41
  42     CASE 2: CHECK FOR SUBROUTINE ERROR (SUBSTRUCTURE STATEMENT FOR "SELECT")
  43        IF AN ERROR OCCURS  (BLOCK INITIATOR "IF")
  44           PRINT AN ERROR MESSAGE  (PASSIVE STATEMENT)
  45        ENDIF
  46
  47     CASE 3: INVOKE ANOTHER SUBROUTINE  (SUBSTRUCTURE STATEMENT FOR "SELECT")
  48        DO ANOTHER SUBROUTINE  (MODULE INVOCATION STATEMENT)----------->(   )
  49        NOTE:  "DO" IS A SYNONYM FOR "CALL"  (PASSIVE STATEMENT)
  50
  51     ENDSELECT  (TERMINATOR STATEMENT "SELECT")
  52
  53 ENDPROCEDURE (MODULE TERMINATOR STATEMENT "PROCEDURE")
```

```
    *************  MODULE REFERENCE TREE  ******
  LN  PAGE
   1     1  EXAMPLE
   2     2  .  SUBROUTINE
   3     *  .  .  ANOTHER
```

```
                              MODULE
                       CROSS REFERENCE LISTING                      PAGE    4
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                            LINE NUMBERS

ANOTHER
    PAGE   2  PROCEDURE SUBROUTINE                       48
EXAMPLE
    PAGE   1  PROGRAM EXAMPLE                             1    4    6
SUBROUTINE
    PAGE   1  PROGRAM EXAMPLE                            18   20
    PAGE   2  PROCEDURE SUBROUTINE                       30   37   42   47   48
```

ACTION TAKEN	DEFINITION NUMBER AND STATEMENT TYPE ENCOUNTERED								
	21 INITIATOR STATEMENT	23 TERMINATOR STATEMENT	24 SUBSTRUCTURE STATEMENT	25 ESCAPE STATEMENT	26 MODULE INVOCATION STATEMENT	16 PASSIVE STATEMENT	35 TEXT DIRECTIVE	37 TITLE DIRECTIVE	46 CONTROL DIRECTIVE
STATEMENT ENTERED IN TABLE OF CONTENTS	←*		←*					←	
ALL NESTED, OPEN STRUCTURES ARE CLOSED WITH ERROR MESSAGES	←*	←	←					←	
NEW PAGE STARTED IN THE OUTPUT FILE	←*							←	
INDENTATION LEVEL DECREASED		←	←						
STATEMENT WRITTEN TO OUTPUT FILE	←	←	←	←	←	←			
INDENTATION LEVEL INCREASED	←		←						
LEFT ARROW (ESCAPE LEVEL INDICATOR) ADDED TO THE OUTPUT FILE				←					
RIGHT ARROW (CALL INDICATOR) ADDED TO THE OUTPUT FILE					←				
SUBSEQUENT INPUT LINES ARE DIVERTED TO A HOLDING BUFFER							←	←	
THE LINES IN THE HOLDING BUFFER ARE WRITTEN TO THE OUTPUT FILE (BOXED IN BY "*")									←
SUBSEQUENT INPUT LINES ARE DIVERTED BACK FOR NORMAL PROCESSING									←
CONTROL PARAMETERS OF THE SDDL PROCESSOR ARE ALTERED									←

*FOR MODULES ONLY

Fig. 2-1. SDDL Processor Actions

SECTION III

SDDL METHODOLOGY

The following discussion of techniques and styles is intended as a guideline or list of suggestions for using the capabilities of the SDDL language and processor to fullest advantage in striving for the goal of an informative and communicative Software Design Document.

The reader is encouraged to examine these suggestions with a critical eye. Accept what is useful, adapt to your own requirements and taste, and invent new methods, but always keep in mind that the primary purpose of the Software Design Document is to communicate information to other people.

A. USES OF THE SOFTWARE DESIGN DOCUMENT

Throughout the development of the software design, the SDD always represents the definitive word on the current status of the ongoing, dynamic design development process. It is easily updated and readily accessible, in a familiar, informative, readable form, to all members of the development team. This makes the SDD an effective instrument for reconciling misunderstandings and disagreements in the evolutionary development of design specifications, engineering support concepts, and the software design itself. Using the SDD to analyze the design makes it possible to eliminate many errors which otherwise might not be detected until coding is attempted.

As a project management aid, the SDD is very useful for monitoring progress and for recording task responsibilities. It is also effective for analyzing and documenting existing programs.

B. REPRESENTATION OF DATA STRUCTURES

A thorough knowledge of the content and organization of its input and output data is an essential prerequisite to understanding a program. For this reason, much attention was focused on developing data structure representations that effectively display data organization and content. SDDL techniques that facilitate achieving this goal include:

(1) Group the data into appropriate data description modules located in the beginning pages of the SDD.

(2) Provide descriptive names for variables.

(3) Use the period (.) (because it lies low on the printed line and does not interfere with readability) to connect the words of a descriptive phrase to form identifiers which can be automatically displayed in a cross reference table.

(4) Use the underscore to connect the words of a descriptive phrase to form module names.

(5) Use the single or double quote mark to identify single word variable names for cross referencing.

(6) Include information about the data (e.g., units, mode, dimension, etc.) in the data structure module.

(7) Group all data which describe attributes of a design entity with the entity they describe, and provide an entity name which can be used as a qualifier with the attribute.

(8) If the program is to be implemented in a language that does not permit the use of descriptive variable names, include the name to be used in the program code in the data structure.

(9) Define suitable keywords as block initiators to provide automatic indentation. Use the #TERMINATE directive to terminate the data structure blocks without printing a termination statement.

Example: Data Structure

```
PROGRAM VEHICLE_COMPONENTS DATA STRUCTURE

  ENTITY     ENGINE:
       PCT.PEDAL                       [PCTPED]     PERCENT
       'RPM'                           [ENGRPM]     REV/MIN
       'TORQUE'                        [TORQUE]     FT*LB
       MIN.TORQUE                      [MINTOR]     FT*LB
       MAX.TORQUE                      [MAXTOR]     FT*LB
       'HORSEPOWER'     (VECTOR)       [HPOWER]     HP

ENDPROGRAM VEHICLE_COMPONENTS DATA STRUCTURE
```

C. REPRESENTATIONS OF CONTROL/PROCEDURAL STRUCTURES

The constructs of Structured Programming, such as modules (e.g., PROGRAM - RETURN - ENDPROGRAM), iterations (e.g., LOOP - CYCLE/EXITLOOP - REPEAT), conditionals (e.g., IF - ELSE - ENDIF), and selections (e.g., SELECT - CASE - ENDSELECT) are used in a similar manner for software design. The difference is that for software design the structures should convey human-oriented, natural language information to the level of precision and completeness necessary to communicate the design, but free of the syntax constraints and detailed information requirements imposed by programming languages.

Example: Module and block structures, high-level statements

```
 1    PROGRAM MAIN ROUTINE
 2       LOOP UNTIL THERE IS NO MORE DATA
 3          READ THE DATA AND CHECK IT
 4          IF THE DATA IS BAD OR INCOMPLETE
 5       <-----CYCLE TO THE NEXT CASE
 6          ELSE
 7             CALL DATA_PROCESSING ROUTINE---------> (9)
 8          ENDIF
 9       REPEAT
10       TERMINATE THE PROGRAM
11    ENDPROGRAM
```

If the design must specify a list of conditions where all must be tested and acted upon if true (in contrast to the SELECT-CASE-ENDSELECT construct, which finds and executes only the first true condition), a new structure is recommended in place of a sequence of IF-ENDIF structures. Use the #DEFINE directive to establish the following structure:

CHECK - block initiator
ENDCHECKLIST - block terminator
CONDITION - substructure

Example: Checklist

As input:

```
#DEFINE BLOCK CHECK,  ENDCHECKLIST,, CONDITION

PROGRAM FOR VACATION PREPARATION

CHECK AND ACT ON ALL TRUE CONDITIONS IN THE FOLLOWING LIST

CONDITION: CAR NEEDS TO BE SERVICED
TAKE CAR TO THE SERVICE STATION
GET GAS AND OIL
INFLATE TIRES

CONDITION: DELIVERIES HAVE TO BE CANCELLED
CANCEL NEWSPAPER
CANCEL MILK

CONDITION: TRIP HAS TO BE PLANNED
GET MAPS
MAKE HOTEL RESERVATIONS

ENDCHECKLIST
ENDPROGRAM
```

As output:

```
PROGRAM FOR VACATION PREPARATION

   CHECK AND ACT ON ALL TRUE CONDITIONS IN THE FOLLOWING LIST

   CONDITION: CAR NEEDS TO BE SERVICED
      TAKE CAR TO THE SERVICE STATION
      GET GAS AND OIL
      INFLATE TIRES

   CONDITION: DELIVERIES HAVE TO BE CANCELLED
      CANCEL NEWSPAPER
      CANCEL MILK

   CONDITION: TRIP HAS TO BE PLANNED
      GET MAPS
      MAKE HOTEL RESERVATIONS

   ENDCHECKLIST
ENDPROGRAM
```

The following forms are recommended for use when the design has progressed to the point where engineering calculations need to be expressed:

Example: Calculation - Equation not yet determined

```
CALCULATE VEHICLE.STATE: DISTANCE.TRAVELLED (TARGETTED)
* GIVEN: VEHICLE.STATE: DISTANCE.TRAVELLED (CURRENT)
*        VEHICLE.STATE.VELOCITY (CURRENT)
*        VEHICLE.STATE.ACCELERATION (TARGETTED)
*        TIME INCREMENT
```

Example: Calculation - Equation included

```
COMPUTE VEHICLE.STATE: DISTANCE.TRAVELLED (TARGETTED) =
        D + V*T + (A/2)*T**2
   D == VEHICLE.STATE: DISTANCE.TRAVELLED (CURRENT)
   V == VEHICLE.STATE: VELOCITY (CURRENT)
   T == TIME.INCREMENT
   A == VEHICLE.STATE: ACCELERATION (TARGETTED)
```

Indentation in the examples above may be imposed by indenting the input statements or by defining COMPUTE to be a Block Initiator keyword.

D. SPECIFICATION OF MODULE INTERFACES

Explicit specification of the data passed between modules and accessed from a global store will eliminate many debugging problems in the coding and integration stages.

(1) Use the words GIVEN and YIELD to specify parameters transmitted to and returned from a module. Use the word USING to specify global variables accessed.

(2) List the GIVEN and YIELD parameters with Module Invocation statements.

Example: Display of module interface parameters

```
NOW CALCULATE_DRIVE_WHEEL OUTPUT_REQUIRED-------------------------> ( 38)
* GIVEN: VEHICLE.STATE:
*        SCHEDULED.TIME
* YIELD: VEHICLE.STATE: TIRE.RPM, ACCELERATION
*        WHEEL FORCE REQUIRED
*        WHEEL TORQUE REQUIRED
```

In the above example, NOW is the Module Invocation keyword. The lines specifying arguments passed to and from the module all begin with an asterisk to emphasize their association with the Invocation statement.

(3) List USING, GIVEN, and YIELD parameters with Module Initiator statements.

Example: Display of parameters with the module definition

```
PROCEDURE TO CALCULATE_DRIVE_WHEEL_OUTPUT_REQUIRED
   **************************************************
   *                                                *
   *  USING: DRIVE.POWER.TRAIN: DATA                *
   *         CHASSIS: DATA                          *
   *  GIVEN: VEHICLE.STATE:                         *
   *         SCHEDULED.TIME                         *
   *  YIELD: VEHICLE.STATE: TIRE.RPM, ACCELERATION  *
   *         WHEEL FORCE REQUIRED                   *
   *         WHEEL TORQUE REQUIRED                  *
   *                                                *
   **************************************************
```

The parameters in this structure are set off by using the #TEXT - #END directives to enclose them in a box formed by asterisks. In addition to the GIVEN and YIELD arguments, the USING category lists global parameters which are accessed by the module.

E. INCLUSION OF MANAGEMENT INFORMATION IN THE SDD

Project management information, just as program design, must be kept up to date and accurate. The SDD is the ideal place to maintain this information, and the language can be used effectively to present the information. Listed below are several Module Initiator statements which suggest kinds of management information, as indicated by their wording, that should be included in the SDD.

```
PROGRAM OBJECTIVES
PROGRAM REVISIONS MEMORANDA
PROGRAM MEETING CALENDAR & AGENDA
PROGRAM DOCUMENT READING CONVENTIONS
PROGRAM COMPLETION SCHEDULE
```

F. ADDITIONAL USES OF THE CROSS REFERENCE CAPABILITY

The SDD typically will contain much information, in addition to the names of design parameters, for which it would be useful to have a cross reference. Individual cross reference tables for each type of information can be obtained by associating a different cross reference title with each (see the #MARK directive). Some that have proved to be useful appear in the sample design which follows. The example shows the form of the #MARK directive which establishes the cross reference character and the way in which the data appear in the main body of the SDD. The pound sign (#) has been used in the input to cause some information to be printed at the right-hand margin of the SDD for increased readability (See Section IV, 1.6, PASSIVE STATEMENT, item 5).

Example: Uses of the cross reference capability

As input:

```
#MARK REVISIONS % FOOTNOTES [ FILE NAMES $
#MARK UPDATE RESPONSIBILITY ?
PROGRAM TO PROCESS CUSTOMER DATA # [REF1]
READ NAMES FROM CUSTOMER$FILE # %1
MATCH NAMES TO CREDIT DATA # ?HK
WRITE CREDIT INFO TO CREDIT$FILE # %2
ENDPROGRAM
```

As output:

```
                               TABLE OF CONTENTS                          PAGE    I
 PAGE   LINE ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   1      3    PROGRAM TO PROCESS CUSTOMER DATA                              [REF1]

   2         MODULE REFERENCE TREE

   3         MODULE - CROSS REFERENCE LISTING

   4         REVISIONS - CROSS REFERENCE LISTING

   5         FOOTNOTES - CROSS REFERENCE LISTING

   6         FILE NAMES - CROSS REFERENCE LISTING

   7         UPDATE RESPONSIBILITY - CROSS REFERENCE LISTING
```

```
LINE                                                                      PAGE    1
   3 PROGRAM TO PROCESS CUSTOMER DATA                                        [REF1]
   4    READ NAMES FROM CUSTOMER$FILE                                            %1
   5    MATCH NAMES TO CREDIT DATA                                              ?HK
   6    WRITE CREDIT INFO TO CREDIT$FILE                                         %2
   7 ENDPROGRAM
```

```
                                     REVISIONS
                              CROSS REFERENCE LISTING                     PAGE    4
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

%1
    PAGE    1  PROGRAM TO PROCESS                       4
%2
    PAGE    1  PROGRAM TO PROCESS                       6
```

```
                                     FOOTNOTES
                              CROSS REFERENCE LISTING                     PAGE    5
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

[REF1
    PAGE    1  PROGRAM TO PROCESS                       3
```

```
                                       FILE NAMES
                                CROSS REFERENCE LISTING                       PAGE    6
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                                    LINE NUMBERS

CREDIT$FILE
    PAGE   1  PROGRAM TO PROCESS                                 6
CUSTOMER$FILE
    PAGE   1  PROGRAM TO PROCESS                                 4



                                 UPDATE RESPONSIBILITY
                                CROSS REFERENCE LISTING                       PAGE    7
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                                    LINE NUMBERS

?HK
    PAGE   1  PROGRAM TO PROCESS                                 5
```

SECTION IV

SDDL USER'S REFERENCE GUIDE

Input to the SDDL processor consists of a sequence of design statements and processor control directives.

Statements and Directives begin and end with a line (or record) of the input medium, unless line continuation is explicitly indicated, as described below. Continued lines are concatenated into a single statement for processing.

A. CONTINUATION OF INPUT LINES

A continuation mark, the ampersand, can be used to concatenate several input lines/cards into a single SDDL input statement. The following rules apply to its use:

(1) If the last non-blank character (excluding card sequence numbers -- see #SEQUENCE directive) of an input line is an ampersand, the processor will concatenate the next line of input with the current line to form a single statement.

(2) The ampersand which caused the continuation is removed from the newly formed line, but all other characters, including other ampersands and blanks, are used as they were input to form the new line.

(3) The continuation mark may be used on as many subsequent input lines as desired to form a single SDDL statement or directive.

(4) If the resulting input statement exceeds the allowable output line space, it will be handled as described below.

B. CONTINUATION OF OUTPUT LINES

Occasionally a line of output may be long enough to extend beyond the right-hand page margin. When this occurs, the processor handles the line in the following way:

(1) Beginning at the appropriate indentation level, as many characters (including blanks) of the input line as space permits are printed on the current line.

(2) An ampersand is printed at the right margin.

(3) On the next line of the document, one space to the right of the current indentation level, the remaining characters are printed. Steps 2 and 3 are repeated as many times as necessary to complete the output.

(4) If the indentation level is such that no characters can be printed on the first line, then step 3 is repeated with output beginning at the left margin instead of at the indentation level.

Example: Line continuation (input and output)

As input:

```
1  PRIOR LINE
2  THIS IS AN EXAMPL&
3  E OF A LONG INPUT &
4  LINE & A LONG OUTP&
5  UT LINE
6  NEXT LINE
```

As printed:

```
1  PRIOR LINE
2  THIS IS AN EXAMPLE OF A LONG INPUT LIN &
    E & A LONG OUTPUT LINE
6  NEXT LINE
```

C. SDDL SYNTAX DEFINITION

The SDDL syntax definitions are subdivided into five levels. The primitive definitions are presented in Level 0. Secondary definitions based on the primitive definitions are in level 1. Level 2 contains SDDL statement definitions. The SDDL control directives are defined in level 3. Finally, an overview diagram of an SDDL program, based on definitions in levels 2 and 3, is given in level 4. The definitions in levels 1 through 4 are accompanied by flow diagrams which specify the requirements and options of the syntax. To interpret the diagram, trace the flow line from the term being defined to the end of the definition. Boxes which are unavoidable are requirements, boxes which can be bypassed are options, and boxes which can be returned to are repeatables. The contents of a box may refer to another definition or a literal. To differentiate between them, definitions appear in smaller type, with the definition number in the lower right-hand corner, and literals, in larger type, have no accompanying number.

Primitive Definitions (Level 0)

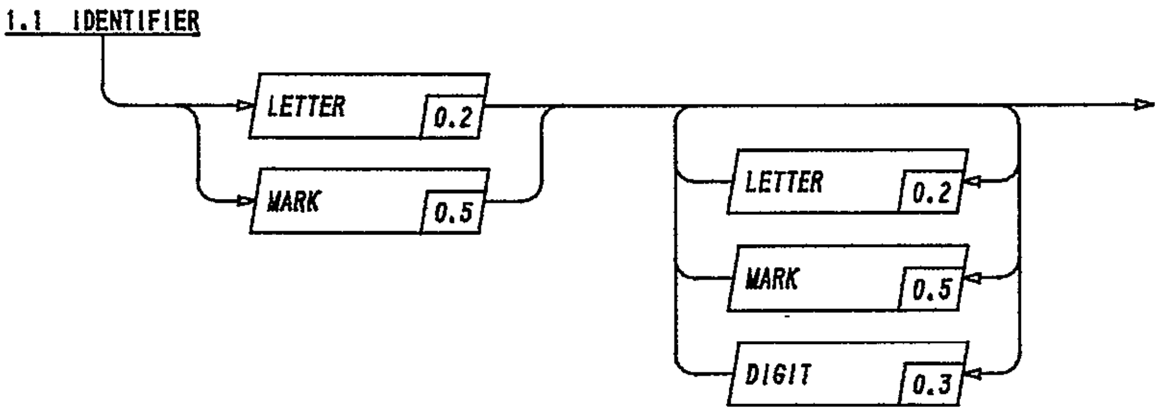
The following description and discussion of SDDL is based on the short list of primitive definitions shown in Table 4-1. Note especially that the definition of a letter includes the pound sign in addition to the alphabet. Also note that initially no MARK characters are defined. As will be explained later in the discussion of the #MARK directive, any punctuation may be converted to a MARK by user specification.

Table 4-1. SDDL Primitive Definitions

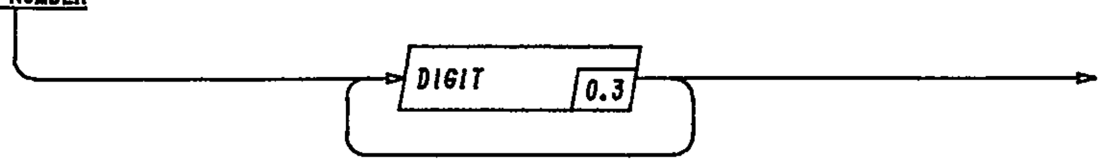
Definition Number	Name	Description
0.1	character set	The entire set of allowable characters (including the blank).
0.2	letter	The alphabet (A-Z) and the pound sign (#).
0.3	digit	The digits (0-9).
0.4	punctuation	The characters remaining after subtracting letter, digit, and the blank from the entire character set.
0.5	mark	Any punctuation which has been converted by a control directive. (Initially, this is the empty set.)
0.6	e.o.s.	The end of an input statement or directive, determined by the end-of-line/record indicator (e.g., carriage return) of an input line without a continuation mark.

1. Secondary Definitions (Level 1)

The definitions of identifier, number, and word shown below are based on the SDDL primitive definitions shown in Table 4-1.

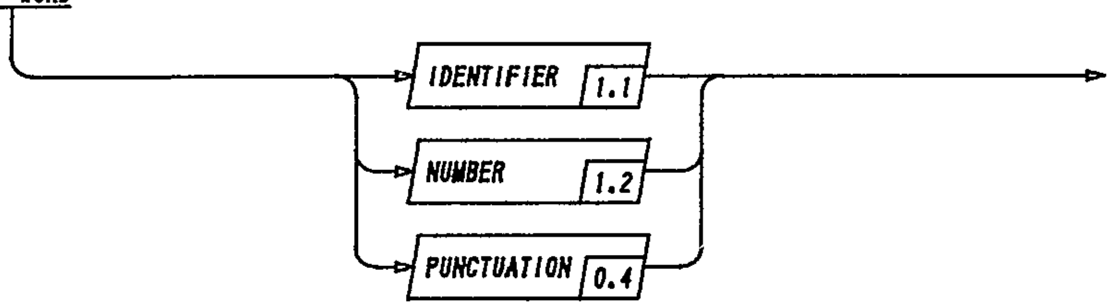


1.2 NUMBER



Note that a number may not have a decimal point. This constraint only affects SDDL control directives which only use integers and has no impact on the design statements which appear in the SDD.

1.3 WORD



As shown above, a word can be an identifier, a number, or punctuation; in short, any token or object definable under the preceding definitions of the language. As in natural languages, the space or blank is a very important part of the syntax which is needed for delimiting or separating words.

Example: Lexical analysis of identifiers

ABC123 X Y#Z?E 12 4W

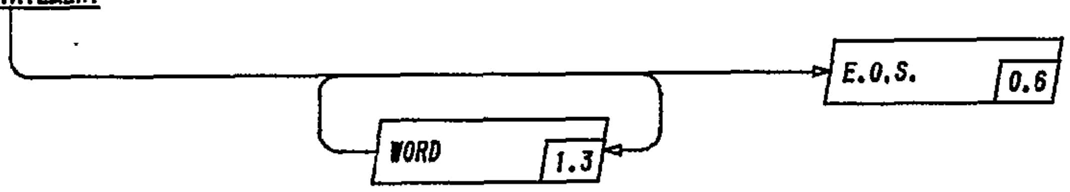
Lexical analysis of the above line yields the following words:

ABC123	(identifier)
X	(identifier)
Y#Z	(identifier)
?	(punctuation)
E	(identifier)
12	(number)
4	(number)
W	(identifier)

If ? had previously been converted to a mark, the result would yield the following words:

ABC123	(identifier)
X	(identifier)
Y#Z?E	(identifier)
12	(number)
4	(number)
W	(identifier)

1.4 STATEMENT



A statement, as shown in the diagram above, consists of any sequence (including the null case) of words.

1.5 KEYWORD

The SDDL processor is keyword-driven. A keyword is an indentifier which has been predefined to be the name of a structure part (initiator, terminator, escape, substructure), a Module Invocation word, or a control directive. Keywords are recognized only in context, i.e., only when they appear as the first word, though not necessarily starting in the first column, of the statement or directive.

1.6 PASSIVE STATEMENT

A Passive statement is any statement which does not begin with a keyword. Passive statements may be used to convey any design information as desired but they do not have any special meaning to the processor as do the Keyword statements.

Passive statements are processed as follows:

(1) Since Passive statements must be imbedded within a module structure, if one does not already exist, the processor will supply a module, with an error message (see next example).

(2) The entire statement is scanned for the appearance of any identifiers which have been designated for inclusion in the cross reference tables. The means for designating identifiers for inclusion in the cross reference tables are explained under the discussion of the #MARK and the #STRING directives.

(3) The input line number (i.e., the number corresponding to the statement's sequential location in the input medium) is written at the left margin.

(4) The entire statement including all blanks is copied to the SDD output file beginning at the current point of indentation.

(5) If the statement contains a pound sign, the portion of the statement which follows it will all be right shifted so that the last non-blank character lines up at the right margin. The pound sign itself is replaced with a space. This feature has many important applications which are examined under the discussion of the #MARK directive.

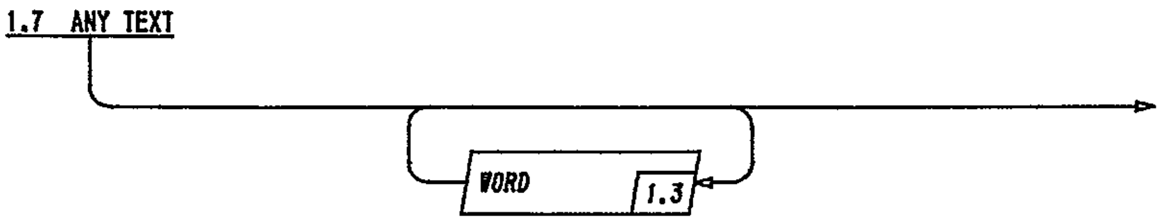
Example: Passive statement without an existing module

As input (input line=1) :

```
ADD 1 # COUNT CASES
```

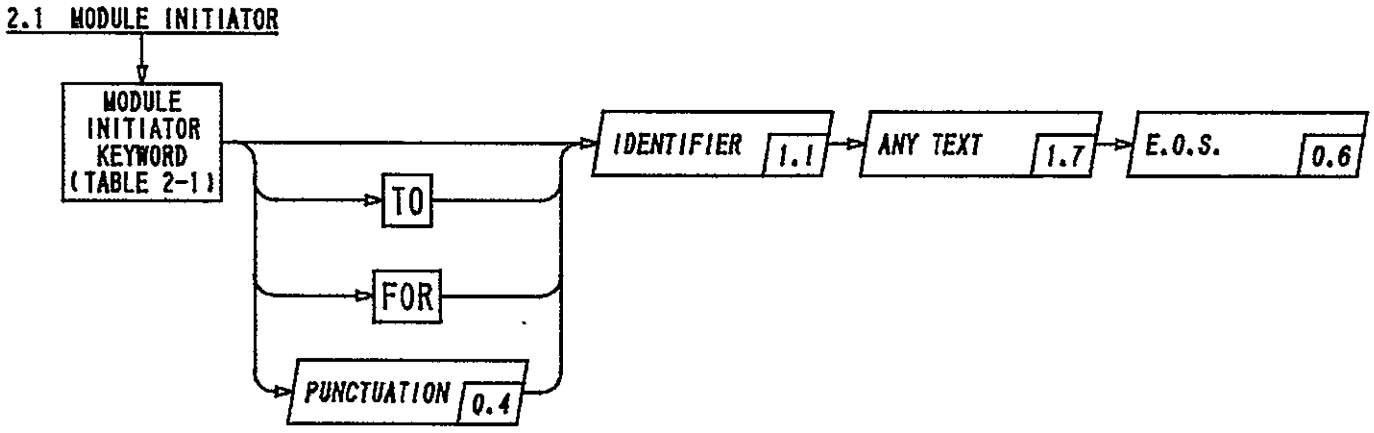
As output:

```
LINE                                         PAGE 1

    PROGRAM UNNAMED# - STATEMENT SUPPLIED BY PROCESSOR

1      ADD 1                             COUNT CASES
```



2. Keyword Statement Definitions (Level 2)

This section describes the Keyword statements which drive the processor formatting actions. The primary function of the processor is to reproduce the input statements on the SDD output file in a manner which enhances the reader's capabililty to understand the resulting document with the least effort. This is accomplished by indentation of statements within structures and superimposition of flow lines to highlight structure escapes and module invocations. The actions taken by the processor in response to specific statement types are described below and summarized in Fig. 2-1.



Example: Module initiator statement

PROGRAM TO READ THE PROGRAM INPUT

(1) The keyword PROGRAM is recognized as a Module Initiator.

(2) The optional noise word TO (FOR or punctuation are alternative noise words) is ignored.

(3) The next identifier, READ, is established as the module's name and recorded for future cross referencing. The remaining words, including the second appearance of PROGRAM, are handled as though they were part of a Passive statement).

(4) Since a module is the highest-level structure and may not be nested within other structures, the processor terminates any open structures (i.e., structures which have been initiated but left unterminated) with appropriate error messages.

(5) The entire Module Initiator statement is entered into the SDD table of contents.

(6) The module structure is entered into a push-down (last-in, first-out) structure stack for later matching with subsequent statements specifying other parts of the structure.

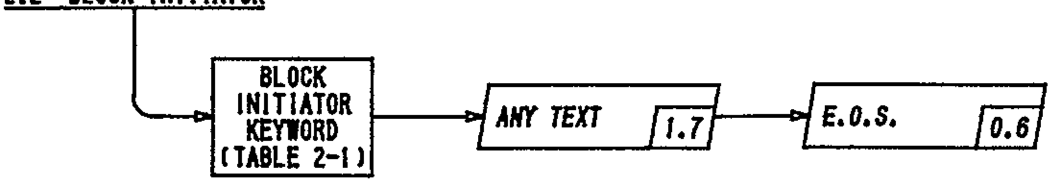
(7) A new page of the SDD is started with appropriate heading.

(8) The indentation point is set to level zero (just to the right of the location of the input line number field).

(9) The statement is written to the SDD output file in the manner described above for Passive Statements.

(10) The indentation is increased one level by moving the indentation point the required number (default = 3) of spaces to the right.

2.2 BLOCK INITIATOR



Example: Block initiator statement

LOOP UNTIL FILES A, B & C HAVE BEEN READ

(1) The keyword LOOP is recognized as a Block Initiator keyword.

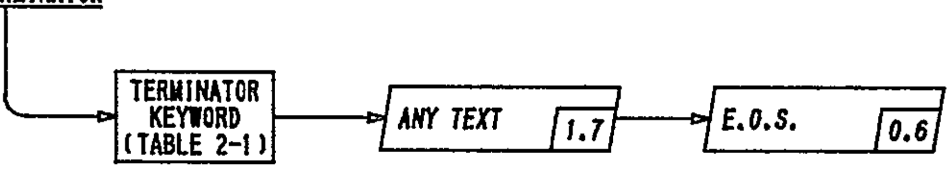
(2) Since blocks must be nested within modules, if an open module does not already exist, the processor supplies a module initiator statement and an error message.

(3) The block structure is entered into a push-down (last-in, first-out) structure stack for later matching with subsequent statements specifying other parts of the structure.

(4) The statement is written to the SDD output file, as described above for Passive statements.

(5) Indentation is increased one level by moving the indentation point the required number (default = 3) of spaces to the right.

2.3 TERMINATOR



Example: Terminator statement

ENDPROGRAM TO READ INPUT

(1) The identifier ENDPROGRAM is recognized as a Terminator keyword.

(2) The structure stack is searched for a matching Structure Initiator. If none is found, the statement is processed as a Passive statement and is followed by an error message. No further action is taken.

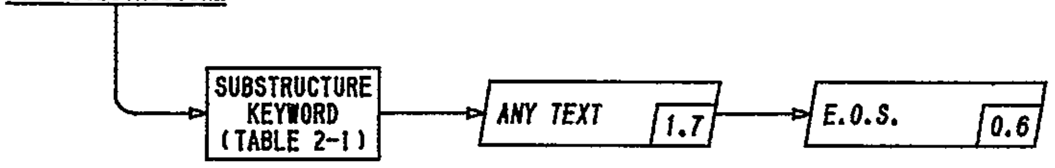
(3) If a matching structure is found, all nested open structures are terminated with error messages.

(4) The structure to be terminated is removed from the top of the structure stack.

(5) Indentation is decreased (shifted left) to match the indentation of the Structure Initiator statement.

(6) The statement is written to the SDD output file in the manner of a Passive statement.

2.4 SUBSTRUCTURE



Example: Substructure statement

ELSE TRY ANOTHER ALTERNATIVE

(1) The identifier ELSE is recognized as a Substructure keyword.

(2) The structure stack is searched for a matching Structure Initiator. If none is found, the statement is processed as a Passive statement and followed with an error message. No further action is taken.

(3) If a matching structure is found, all intervening, open structures are terminated with error messages.

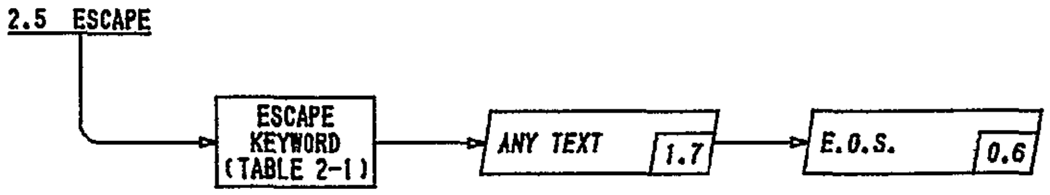
(4) In the case where the substructure corresponds to a module (rather than a block) the statement is entered into the SDD table of contents.

(5) Indentation is decreased (shifted left) to match the indentation of the Structure Initiator statement.

(6) The statement is written like a Passive statement.

(7) Indentation is increased one level (shifted right), as when the structure had just been initiated, in effect re-initiating the structure.

2.5 ESCAPE



Example: Escape statement

EXITLOOP IF DELTA < EPSILON

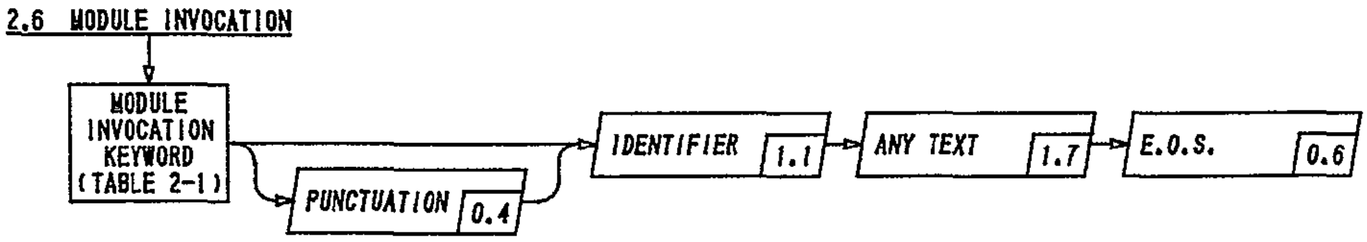
(1) The identifier EXITLOOP is recognized as an Escape keyword.

(2) The statement is written to the SDD in the manner described for the Passive statement.

(3) The structure stack is searched for a matching Structure Initiator. If none is found, an error message is added to the SDD output file.

(4) If a matching structure is found, the escape statement is completed by the addition of a flow line (left arrow) extending from the current indentation level to the indentation level of the matching Structure Initiator statement.

2.6 MODULE INVOCATION



Example: Module invocation statement

CALL : INITIALIZATION ROUTINE

(1) The identifier CALL is recognized as a Module Invocation keyword.

(2) The optional punctuation, :, is ignored.

(3) The identifier INITIALIZATION is established as the name of the module to be invoked and recorded for module cross referencing.

(4) The statement is written to the SDD in the manner described for a Passive statement.

(5) The output line is augmented by a flow line (right arrow) extending from the rightmost non-blank character of the statement to within six columns of the right-hand margin.

(6) The last six columns of the output line are filled in with parentheses enclosing the page number of the module referenced by the Module Invocation statement.

The processor actions for SDDL statements described above are summarized in Figure 2-1. The following example illustrates the statements as they might be combined in a simple design:

Example: A simple design

As input:

```
PROGRAM TO SUMMARIZE DATA
CALL INITIALIZE
LOOP UNTIL ALL NUMBERS HAVE BEEN READ
READ A VALUE
CALL ERRORCHECK
IF THE ERRORCHECK INDICATES AN ERROR
PRINT THE FOLLOWING MESSAGE
   "SOMETHING'S WRONG"
CYCLE BACK FOR ANOTHER ITERATION
ELSE
SUM VALUES & SQUARED VALUES
INCREMENT COUNTER
ENDIF
REPEAT
DISPLAY MEAN AND STANDARD DEVIATION.
ENDPROGRAM
PROCEDURE TO INITIALIZE
   VARIABLE                         INITIAL VALUE
   SUM                                   0.0 #REAL
   SUM OF SQUARES                        0.0 #REAL
   COUNT                                 0 #INTEGER
   LOWER BOUND                           0 #REAL
   UPPER BOUND                         100.0 #REAL
PROCEDURE FOR ERRORCHECK
INITIALIZE ERRORCHECK TO INDICATE AN ERROR
IF LOWER BOUND < VALUE
IF VALUE < UPPER BOUND
RESET ERRORCHECK TO INDICATE NO ERROR
ENDIF
ENDIF
```

As output:

```
                              TABLE OF CONTENTS                          PAGE    I
 PAGE   LINE +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   1       1    PROGRAM TO SUMMARIZE DATA
   2      17    PROCEDURE TO INITIALIZE
   3      24    PROCEDURE FOR ERRORCHECK

   4          MODULE REFERENCE TREE

   5          MODULE - CROSS REFERENCE LISTING
```

```
LINE                                                                     PAGE    1
   1 PROGRAM TO SUMMARIZE DATA
   2    CALL INITIALIZE--------------------------------------------------->(  2)
   3    LOOP UNTIL ALL NUMBERS HAVE BEEN READ
   4       READ A VALUE
   5       CALL ERRORCHECK------------------------------------------------>(  3)
   6       IF THE ERRORCHECK INDICATES AN ERROR
   7          PRINT THE FOLLOWING MESSAGE
   8             "SOMETHING'S WRONG"
   9    <-----CYCLE BACK FOR ANOTHER ITERATION
  10       ELSE
  11          SUM VALUES & SQUARED VALUES
  12          INCREMENT COUNTER
  13       ENDIF
  14    REPEAT
  15    DISPLAY MEAN AND STANDARD DEVIATION
  16 ENDPROGRAM
```

```
LINE                                                                     PAGE    2
  17 PROCEDURE TO INITIALIZE
  18       VARIABLE                     INITIAL VALUE
  19       SUM                               0.0                              REAL
  20       SUM OF SQUARES                    0.0                              REAL
  21       COUNT                             0                             INTEGER
  22       LOWER BOUND                       0                                REAL
  23       UPPER BOUND                     100.0                              REAL
     ENDPROCEDURE - STMT SUPPLIED BY PROCESSOR
```

```
LINE                                                                     PAGE    3
  24 PROCEDURE FOR ERRORCHECK
  25    INITIALIZE ERRORCHECK TO INDICATE AN ERROR
  26    IF LOWER BOUND < VALUE
  27       IF VALUE < UPPER BOUND
  28          RESET ERRORCHECK TO INDICATE NO ERROR
  29       ENDIF
  30    ENDIF
     ENDPROCEDURE - STMT SUPPLIED BY PROCESSOR
```

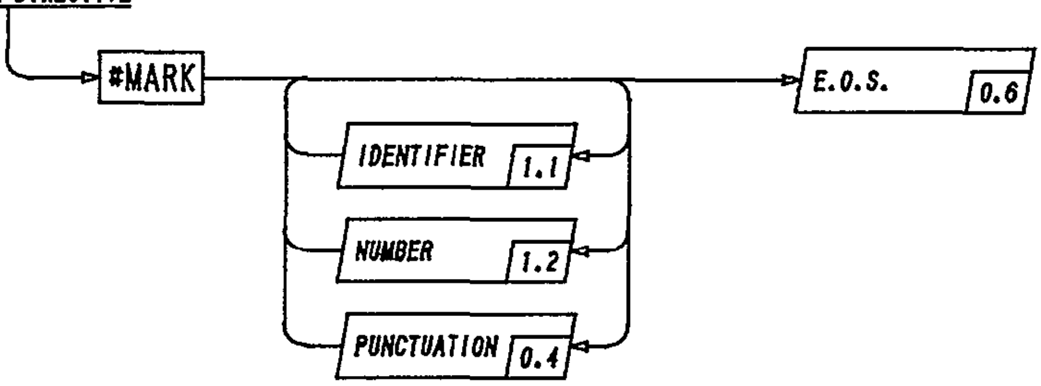
```
   *************  MODULE REFERENCE TREE  ******                    PAGE    4
 LN  PAGE
  1     1  SUMMARIZE
  2     2  .  INITIALIZE
  3     3  .  ERRORCHECK


                                 MODULE
                         CROSS REFERENCE LISTING                   PAGE    5
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

ERRORCHECK
    PAGE   1  PROGRAM TO SUMMARIZE                     5    6
    PAGE   3  PROCEDURE FOR ERRORCHECK                24   25   28
INITIALIZE
    PAGE   1  PROGRAM TO SUMMARIZE                     2
    PAGE   2  PROCEDURE TO INITIALIZE                 17
    PAGE   3  PROCEDURE FOR ERRORCHECK                25
SUMMARIZE
    PAGE   1  PROGRAM TO SUMMARIZE                     1
```

3. Control Directives (Level 3)

Control directives allow the user to set processor background control specifications (e.g., page width, indentation) and to cause some immediate actions to be taken (e.g., page eject). Control directives are read, interpreted, and acted upon by the processor. They are not written to the SDD output file and hence are not seen in the final document. Control specifications set by directives are put into effect as soon as they are interpreted and remain in effect for all subsequent input, or until overridden by another directive. Directives can be used to set and reset processor control specifications as often as desired. The SDDL control directives are defined and described on the following pages. The sequence of presentation is intended to avoid lookahead caused by definitions based on terms defined on subsequent pages.

Control directive keywords all begin with the pound sign character. They are preset (see Table 2-2) and must not be altered. The user must be careful not to define a new meaning for a control directive keyword (see #DEFINE directive) since it will cause the preset definition to be overridden and lost.

3.1 MARK DIRECTIVE



Selection of words or identifiers for cross referencing is controlled by the #MARK and the #STRING directives. When using the #MARK directive, the designer specifies a list of punctuation symbols which the processor will subsequently treat in the following manner:

(1) All punctuation appearing in the statement is converted into a MARK (syntax definition 0.5), i.e., those characters which are used to form identifiers. They can then be used as connectors to build a single identifier out of separate words.

Example: Mark directive without cross reference title

```
#MARK .
EVERY.GOOD.BOY DOES FINE
```

(2) Every identifier which includes a MARK, such as in EVERY.GOOD.BOY in the example above, is included in a cross reference listing produced at the end of the design document.

Titles for the cross reference listings may be supplied by placing any string of characters (except punctuation) prior to the punctuation to be converted. If, as in the above example, no title is supplied prior to the first punctuation in the directive, a blank title is assumed.

The SDDL processor provides individual cross reference listings for each unique title found in the #MARK and/or #STRING directives. Identifiers containing MARKs which were specified with identical titles are merged into a single cross reference listing. Titles are considered to be identical if, after deleting leading and following blanks, they are an exact, character-by-character match, including internal (between word) blanks. Identifiers which contain marks associated with several unique titles will appear in each appropriate cross reference. These conventions are exemplified below, and an additional, more comprehensive example is given following the #STRING directive.

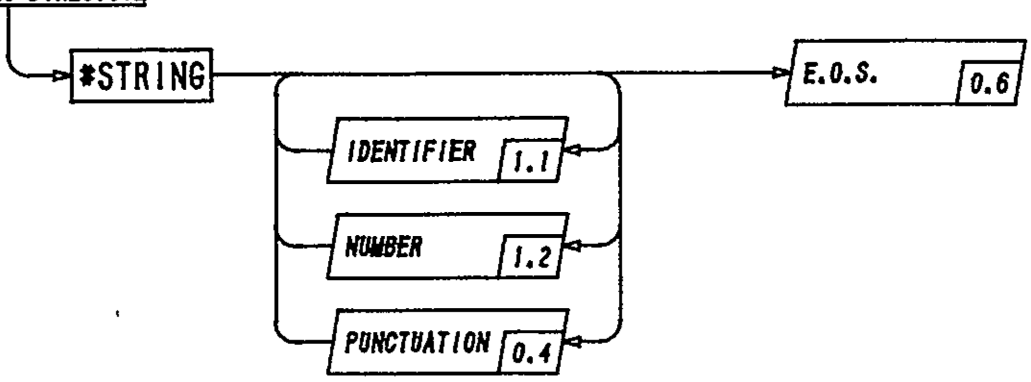
Example: Mark directive with and without cross references titles

```
#MARK     ?! DATA ITEMS %  REVISIONS $

#MARK     ; DATA ITEMS .:
```

The MARKs specified in the above example are associated with the titles (null), DATA ITEMS, and REVISIONS as follows:

```
           (null)
   CROSS REFERENCE LISTING

           ?  !  ;


         DATA ITEMS
   CROSS REFERENCE LISTING

           %  .  :


          REVISIONS
   CROSS REFERENCE LISTING

              $
```

3.2 STRING DIRECTIVE



This directive allows the user to specify one or more punctuation marks to be used as string delimiters. The purpose of enclosing text within string delimiters is to have it included in a cross reference table at the end of the document. The following rules govern the use of this feature.

(1) Several punctuation symbols may be specified as string delimiters but no distinction is made between left (opening) or right (closing) delimiters

Example: String directive with 2 delimiters specified

```
#STRING ()
1     SAMPLE STATEMENT (STRING   ONE(
2     ) STRING TWO (NOT A STRING) STRING ABC)
```

In the above example, the following text segments are defined and will be cross referenced:

```
"STRING   ONE"    "STRING TWO"    "STRING ABC"
```

(2) Preceding and following blanks are excluded from the string, but interior blanks are included.

Example: String directive - internal and external blanks

```
#STRING '
LINE 1         '  ABC D'
LINE 2         'ABC D  '
LINE 3         'ABC   D'
```

The strings in LINE 1 and LINE 2 are the same because they match exactly after preceding and following blanks are stripped off. The string in LINE 3 does not match the others because it does not have the same number of spaces between ABC and D. Each unique string, where uniqueness is defined by rules 1 and 2, becomes a single entry in the cross reference.

(3) If the closing delimiter is omitted, the string is terminated by the end of the input statement.

Example: String directive - missing terminator

```
#STRING '
LINE 1 'ABC' AND 'DEF G
```

Strings ABC and DEF G are recognized.

(4) If the text enclosed in string delimiters consists of a single identifier, regardless of preceding or following blanks, it is recognized as described above, but in addition, the processor will thereafter recognize and cross reference the named identifier whether it appears with or without string delimiters.

Example: Strings containing a single identifier

```
#STRING "
LINE 1          "VEHICLE "
LINE 2          VEHICLE AND VEHICLE
```

In the above example, VEHICLE is recognized and the cross reference will show that it was found once in LINE 1 and twice in LINE 2.

(5) A title for the cross referencing of text strings may be supplied by including any characters except punctuation between the #STRING keyword and the first punctuation symbol to be converted to a string delimiter.

The title, including (null), supplied with the #STRING directive is compared with the titles supplied with the #MARK directives for merging of the cross reference listings. When several #STRING or #MARK directives with conflicting title specifications are used, the rule followed is that the last usage overrides all prior usage.

An execution-time option (N-option) provides a means to suppress the output of the cross reference table which has the null title.

Example: Mark and String directives

As input:

```
  1       .#MARK ?;  DATA ITEMS %     REVISIONS  $
  2        #MARK DATA ITEMS .:
  3        #STRING DATA ITEMS "
  4        PROGRAM TO READ DATA AND "CHECK" IT
  5        READ VEHICLE:  ,  MAX.RPM , %POWER , "AND WHAT EVER ELSE THERE IS "
  6        IF ANY VALUES ARE UNKNOWN?  OR UNTESTED?
  7        CHECK THE DATA;;  FOR DOUBTFUL.STUFF?  $1
  8        ENDIF
  9        AN ADDITIONAL CHECK MAY BE NEEDED HERE
 10        ENDPROGRAM
```

As output:

```
                              TABLE OF CONTENTS                          PAGE    I
 PAGE   LINE +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   1      4    PROGRAM TO READ DATA AND "CHECK" IT

   2         MODULE REFERENCE TREE

   3         MODULE - CROSS REFERENCE LISTING

   4         DATA ITEMS - CROSS REFERENCE LISTING

   5         REVISIONS - CROSS REFERENCE LISTING

   6         CROSS REFERENCE LISTING
```

```
LINE                                                                     PAGE    1
   4 PROGRAM TO READ DATA AND "CHECK" IT
   5    READ VEHICLE:  ,  MAX.RPM , %POWER , "AND WHAT EVER ELSE THERE IS "
   6    IF ANY VALUES ARE UNKNOWN?  OR UNTESTED?
   7       CHECK THE DATA;;  FOR DOUBTFUL.STUFF?  $1
   8    ENDIF
   9    AN ADDITIONAL CHECK MAY BE NEEDED HERE
  10 ENDPROGRAM
```

```
                                     DATA ITEMS
                               CROSS REFERENCE LISTING                   PAGE    4
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

%POWER
    PAGE   1  PROGRAM TO READ                           5
AND WHAT EVER ELSE THERE IS
    PAGE   1  PROGRAM TO READ                           5
CHECK
    PAGE   1  PROGRAM TO READ                           4    7    9
DOUBTFUL.STUFF?
    PAGE   1  PROGRAM TO READ                           7
MAX.RPM
    PAGE   1  PROGRAM TO READ                           5
VEHICLE:
    PAGE   1  PROGRAM TO READ                           5
```

SKEL

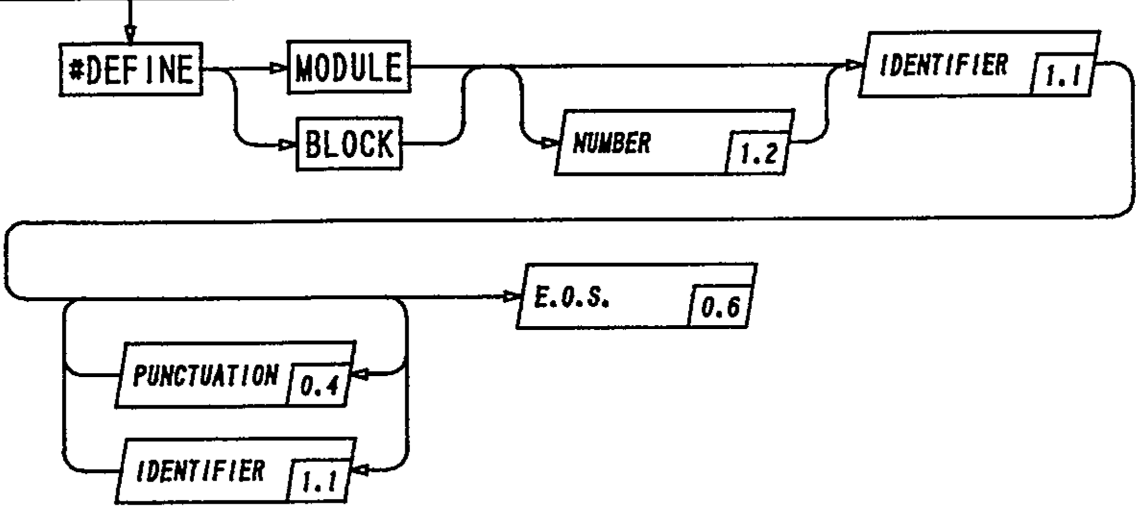
```
                                      REVISIONS
                               CROSS REFERENCE LISTING                   PAGE    5
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

$1
    PAGE   1  PROGRAM TO READ                           7
```

```
                               CROSS REFERENCE LISTING                   PAGE    6
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                          LINE NUMBERS

DATA;;
    PAGE   1  PROGRAM TO READ                           7
DOUBTFUL.STUFF?
    PAGE   1  PROGRAM TO READ                           7
UNKNOWN?
    PAGE   1  PROGRAM TO READ                           6
UNTESTED?
    PAGE   1  PROGRAM TO READ                           6
```

The #DEFINE directive is used to specify new or to delete old SDDL keywords. To select the desired action, one of the four words shown below must follow the SDDL keyword, #DEFINE.

MODULE BLOCK CALL NULL

3.3 DEFINE DIRECTIVE (MODULE, BLOCK)



The word MODULE or BLOCK is used to define a control structure. In SDDL, a control structure has four parts:

(1)	Initiator:	Increases the indentation level for subsequent lines.
(2)	Terminator:	Closes all nested structures left open and returns the indentation level to that of the Initiator statement.
(3)	Escape:	A left arrow is added to the statement to indicate the program control flow. The arrow extends from the indentation level of the escape statement to the indentation level of the corresponding Initiator statement.
(4)	Substructure:	Closes all nested structures left open, returns the indentation level to that of the Initiator statement, prints the line, and increases the indentation level.

When defining a module or block, names for the four parts must be specified in the order shown above. Any punctuation may be used to separate the part names, but care must be taken to avoid using a MARK (i.e., punctuation which has been converted by means of the #MARK or #STRING directive). Names for any of the parts except the initiator may be omitted by using consecutive punctuation to show where a name has been left out. Any text following the name of the substructure will be ignored. Synonyms for part names, except for the initiator name, may be established by additional #DEFINE directives.

Indentation specific to the named structure may be indicated by including an unsigned integer between the word MODULE (BLOCK) and the initiator name. If a zero is specified or the integer is omitted, the current default indentation amount (see #INDENT) will be used.

Example: Three equivalent define directives

```
#DEFINE MODULE 10 PROGRAM, END, STOP, ENTRYPOINT
#DEFINE MODULE 10 PROGRAM`END, STOP ENTRYPOINT
#DEFINE MODULE 10 PROGRAM END STOP ENTRYPOINT WHATEVER
```

type	indentation	initiator	terminator	escape	substructure
module	10	PROGRAM	END	STOP	ENTRYPOINT

Example: Block initiator and terminator definition

```
#DEFINE BLOCK BEGIN END
```

type	indentation	initiator	terminator	escape	substructure
block	default	BEGIN	END		

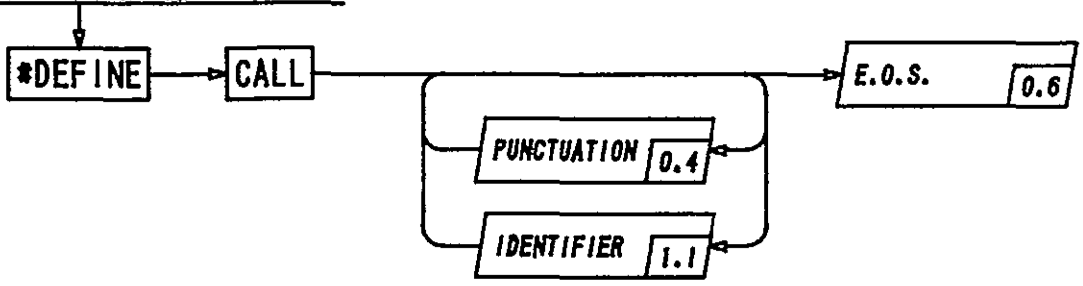
Example: Block definition - escape synonyms

```
#DEFINE BLOCK   START, FINISH, LEAVE
#DEFINE BLOCK   START, , SCRAM
#DEFINE BLOCK 2 START, , VAMOOSE
```

type	indentation	initiator	terminator	escape	substructure
block	2	START	FINISH	LEAVE SCRAM VAMOOSE	---

Note that in this example, the last directive established the indentation amount to be two columns, overriding the default indentation amount indicated on the previous directives.

3.3 DEFINE DIRECTIVE (MODULE INVOCATION)



The word CALL is used with the #DEFINE directive to establish synonyms for the Module Invocation keyword (default keywords are CALL and DO), which indicates that a module is to be invoked at the point where the statement occurs. The identifiers to be defined as synonyms are listed after the word CALL. Punctuation for separating the words is optional.

Example: CALL keyword definitions

```
#DEFINE CALL PERFORM EXECUTE, GOGOGO
#DEFINE CALL DOITNOW
```

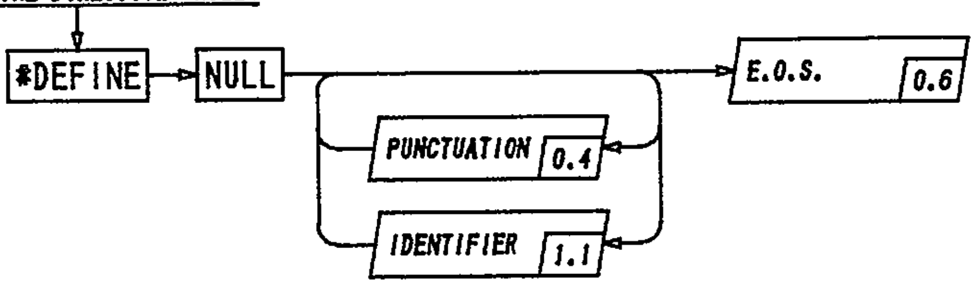
Example: Call keywords with marks

```
#MARK .
#DEFINE CALL DO.IT.NOW, PERFORM
```

The identifier DO.IT.NOW (also PERFORM) becomes a Module Invocation keyword because the period has been converted to a MARK by the prior #MARK directive. Where DO.IT.NOW appears in the context of a keyword (first word of the statement), it will not be included in the cross reference table.

When a Module Invocation statement is encountered, the processor places the statement in the output file with the appropriate indentation and adds a right arrow from the rightmost character in the line to the right margin. Matching parentheses are added to the right of the arrow to provide a place for adding the page number of the called module. If the module that is referenced in the Module Invocation statement has been defined on a prior page, the page number is supplied in the allocated space when the statement is encountered. Page reference numbers which cannot be supplied immediately will be filled in automatically on a second pass over the output file. The user may exercise the P option at execution time to suppress the second pass, which supplies the remaining page reference numbers.

3.3 DEFINE DIRECTIVE (NULL)



The NULL action of this directive provides a means for returning any previously defined keywords to the state of being undefined. Punctuation may be used as a keyword separator if desired. MARKs which have been converted to letters by a previous #MARK or #STRING directive may also be listed for redefinition as punctuation. MARKs being redefined in this manner must have adjacent blanks or punctuation to disassociate them from other text.

Example: Nulling keywords

```
#DEFINE NULL PROGRAM, ENDPROGRAM PROCEDURE
```

The words PROGRAM, ENDPROGRAM, and PROCEDURE are not recognized as keywords in the statements following this directive.

Example: Nulling keywords and marks

```
#MARK .$
#DEFINE NULL DO.IT.NOW $
```

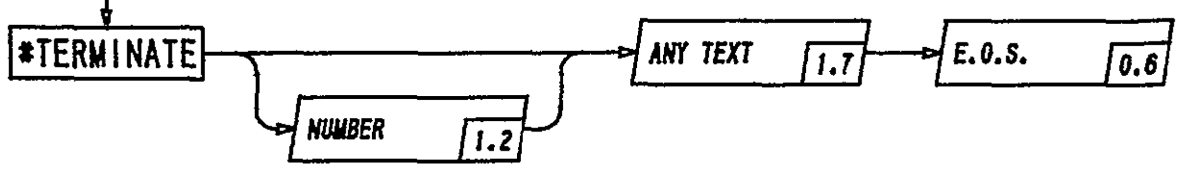
The word DO.IT.NOW is no longer a keyword and $ reverts to punctuation again. The periods in the keyword DO.IT.NOW are part of the identifier (unlike the $ in the example), and therefore the status of the period remains unchanged; i.e., it is still a MARK.

Example: Nulling marks

```
#MARK .
#DEFINE NULL . DO.IT.NOW
```

This example differs in that the status of the period is reconverted to punctuation first and is treated as such in the remainder of the statement. Therefore, DO, IT, and NOW are the words which become undefined. If DO, IT, and NOW are already undefined, they are not affected.

3.4 TERMINATE DIRECTIVE



This directive is a generalized terminator for block structures. It may be used in place of a number of specific terminators (specific terminators must match their respective initiators) to terminate the n innermost, nested, open block structures. If no integer is specified in the directive, only one structure will be terminated. If n is greater than the number of open block structures, they will all be terminated, but the module structure will not be affected.

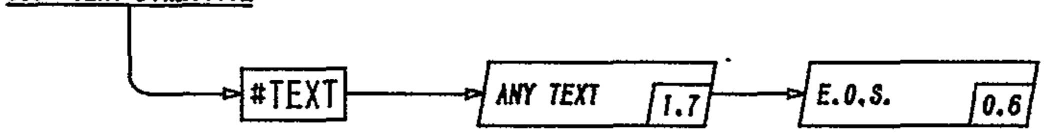
Example: Terminate directive

As input:

```
 1      PROGRAM "TERMINATE" EXAMPLE
 2      IF P INDENT 1 LEVEL
 3      LOOP Q INDENT 1 LEVEL
 4      INDENTATION IS 3 LEVELS DEEP
 5      ENDLOOP - SPECIFIC TERMINATOR
 6      ENDIF - SPECIFIC TERMINATOR
 7      IF P INDENT 1 LEVEL
 8      LOOP Q INDENT 1 LEVEL
 9      INDENTATION IS 3 LEVELS DEEP
10      #TERMINATE 100
11      ALL BLOCK STRUCTURES ARE TERMINATED - MODULE NOT AFFECTED
12      IF P INDENT 1 LEVEL
13      LOOP Q INDENT 1 LEVEL
14      INDENTATION IS 3 LEVELS DEEP
15      #TERMINATE ONLY ONE STRUCTURE TERMINATED
16      IF P INDENT 1 LEVEL
17      INDENTATION IS STILL 3 LEVELS DEEP
18      ENDPROGRAM - STRUCTURES LEFT OPEN ARE TERMINATED BY THE PROCESSOR
```

```
LINE                                                                   PAGE    1
  1 PROGRAM "TERMINATE" EXAMPLE
  2    IF P INDENT 1 LEVEL
  3       LOOP Q INDENT 1 LEVEL
  4          INDENTATION IS 3 LEVELS DEEP
  5       ENDLOOP - SPECIFIC TERMINATOR
  6    ENDIF - SPECIFIC TERMINATOR
  7    IF P INDENT 1 LEVEL
  8       LOOP Q INDENT 1 LEVEL
  9          INDENTATION IS 3 LEVELS DEEP
 11    ALL BLOCK STRUCTURES ARE TERMINATED - MODULE NOT AFFECTED
 12    IF P INDENT 1 LEVEL
 13       LOOP Q INDENT 1 LEVEL
 14          INDENTATION IS 3 LEVELS DEEP
 16       IF P INDENT 1 LEVEL
 17          INDENTATION IS STILL 3 LEVELS DEEP
          ENDIF - STMT SUPPLIED BY PROCESSOR
       ENDIF - STMT SUPPLIED BY PROCESSOR
 18 ENDPROGRAM - STRUCTURES LEFT OPEN ARE TERMINATED BY THE PROCESSOR
```

3.5 TEXT DIRECTIVE



Example: Text directive

#TEXT +

The #TEXT directive is used to signal the beginning of a sequence of lines (not statements) of text intended as commentary to the SDD. When this directive is encountered, the processor performs the following actions:

(1) The first character following the keyword is saved for use in forming a box around the body of text. If no character is specified, the asterisk is used for the boxing character.

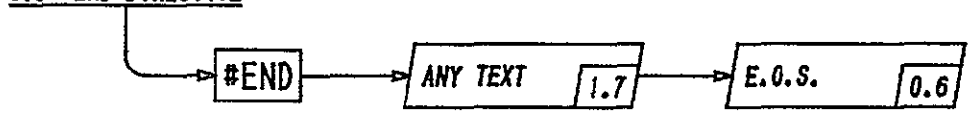
(2) The processor begins reading input lines into a holding buffer and continues until it encounters an input line whose first non-blank character is the pound sign.

(3) The lines buffered in step 2 (this does not include the line which terminated step 2) are not analyzed as statements but simply saved unaltered.

(4) The buffered lines, enclosed in a box formed with the boxing character, are then written to the SDD output file at the current level of indentation.

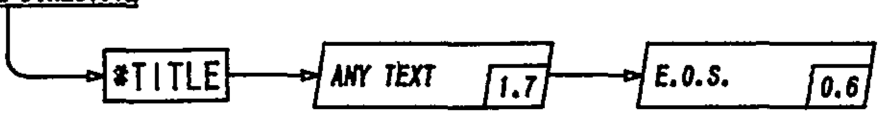
(5) The line which signaled the end of step 2 (the buffering step) is then processed in the usual way. Thus, any control directives or any statement which begins with a pound sign may be used as a terminator and still be recognized for regular processing. If no action other than termination of the text statement is desired, the #END directive may be used.

3.6 END DIRECTIVE



This directive has no effect or purpose other than that of terminating line buffering for #TEXT and #TITLE directives.

3.7 TITLE DIRECTIVE



Example: Title page

#TITLE SDDL DESIGN DOCUMENT

This directive is used to produce a title page in the SDD. The #TITLE directive is similar to the #TEXT directive, but different in that the #TEXT directive is analogous to a Block Initiator statement while the #TITLE directive is analogous to a Module Initiator statement. The processor performs the following actions in response to input of a #TITLE directive.

(1) The keyword #TITLE is recognized.

(2) The initial pound sign is stripped off, and the remainder of the directive is entered into the SDD Table of Contents. Title line entries in the Table of Contents are preceded by a blank line and are written two columns to the left of module entries in order to distinguish them as the beginning of a document section.

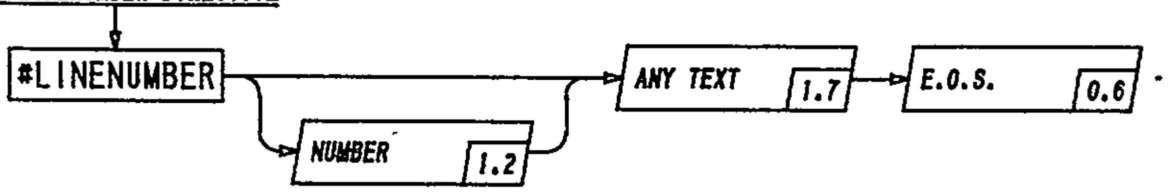
(3) All structures left open are terminated with error messages.

(4) As in the case of a #TEXT directive, the processor reads and buffers input lines until it encounters a line whose first non-blank character is a pound sign. Termination of the title text is the same as for the #TEXT directive.

(5) A new page is started in the SDD output file.

(6) A title page is formed by (a) enclosing the lines in a box formed by asterisks, (b) centering each line within the box, and (c) centering the entire box on the page.

3.8 LINENUMBER DIRECTIVE



This directive provides control of the starting point of the input line numbering sequence which the processor produces in the left margin of the SDD.

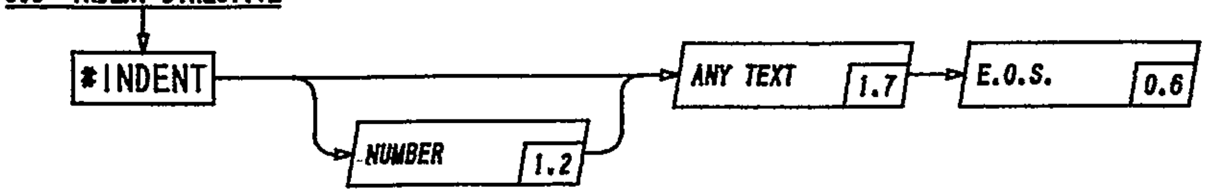
The input line numbers supplied by the SDDL processor correspond exactly to the positional line numbers of the data element (card deck) of the input to the SDDL processor. This feature obviates the need for listing of the raw input for revising and augmenting the SDD. Where more than one element (deck) is used as input to SDDL, it is desirable to reset the line counter so that numbering can be made to match the subsequent elements (card decks.)

If this instruction is issued without an accompanying integer, the processor will begin numbering subsequent lines from 1; otherwise it will begin numbering with the value specified by the integer. The syntax of this directive allows noise to be used for commentary if desired.

Examples: Line number specification

```
#LINENUMBER 1001 STARTS THE NEXT ELEMENT

#LINENUMBER
```

3.9 INDENT DIRECTIVE



The SDDL #INDENT directive allows the user to override the default value for the number of spaces to be skipped for automatic statement indentation.

User-defined structures (see #DEFINE directive 3.3) which do not have a specific indentation amount declared, and SDDL default structure definitions always use the current default indentation value. The initial value of the system defined default indentation amount is three spaces.

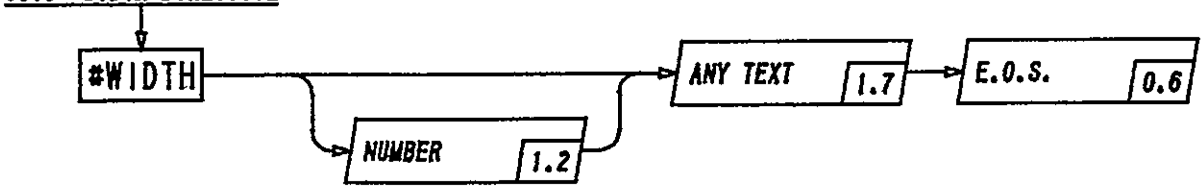
Text following the integer (i.e., noise) may be used for commentary if desired. If no integer is specified in the directive, the default value of three spaces is assumed.

Examples: Indentation specification

#INDENT 5 SPACES UNLESS OTHERWISE SPECIFIED

#INDENT SET TO DEFAULT OF THREE SPACES

3.10 WIDTH DIRECTIVE



The #WIDTH directive provides user control of the width of the output pages. The default page width is 80 characters = 20 cm (8 in.).

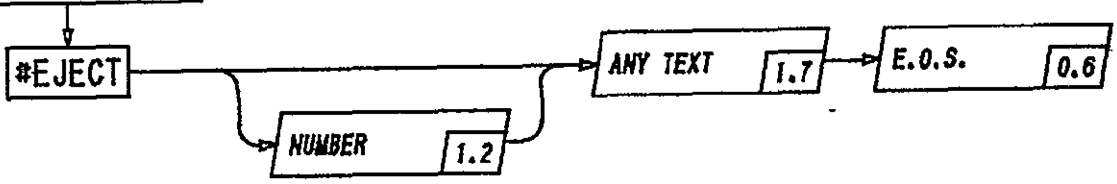
An integer specifying the width, in characters/output line, should be supplied. If the integer value is not in the range 60-130, an error message will be printed and the page width will not be altered. If no integer is specified in the directive, the default value of 80 columns is assumed.

This directive may be used as many times as desired throughout the program. Each use affects only the output which follows it.

Example: Page width specification

```
#WIDTH 130 COLUMNS FOR A TABLE
      .
      .
      .
#WIDTH RESUME NORMAL PAGE WIDTH
```

3.11 EJECT DIRECTIVE



This directive provides immediate control of the start of a new page in the SDD. This page control is over and above the automatic new page start caused by (1) a title, (2) the beginning of a new module, or (3) page overflow. When a module becomes lengthy enough to cause an overflow to a new page, it is often desirable to control the start of the new page to prevent a group of lines from being split over a page boundary.

The #EJECT directive, without an accompanying integer, causes a new page to be started beginning with the next SDDL statement in the input stream.

Examples: Page ejection

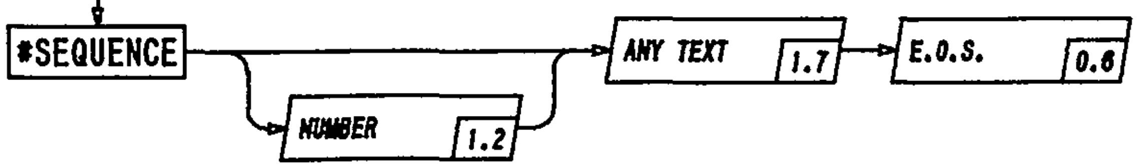
```
#EJECT

#EJECT A PAGE NO MATTER WHAT
```

When an integer is included in this command, it causes a new page to be started only when the remainder of the page cannot accommodate the number of lines specified by the value of the integer. An integer value greater than 50 gives rise to an error message and causes the directive to be ignored. Noise following the integer is ignored and may therefore be used for commentary.

Examples: Conditional page ejection

```
#EJECT 5

#EJECT 7 THE FOLLOWING 7 LINES MUST BE KEPT TOGETHER
```

3.12 SEQUENCE DIRECTIVE



The #SEQUENCE directive is provided for use with card input to the SDDL processor. When SDDL is used in a timesharing environment with file management and editing capabilities, card sequencing is unnecessary. In this case, the full 80 columns of the input line may be used entirely for SDDL statements and directives and the #SEQUENCE directive can be ignored, except to avoid its inadvertent use. The input line numbers supplied in the left margin of the output file correspond exactly to the line to edit in the input file for corrections and updates and may be reliably used for this purpose. This feature makes it unnecessary to punch cards or print out copies of the input file.

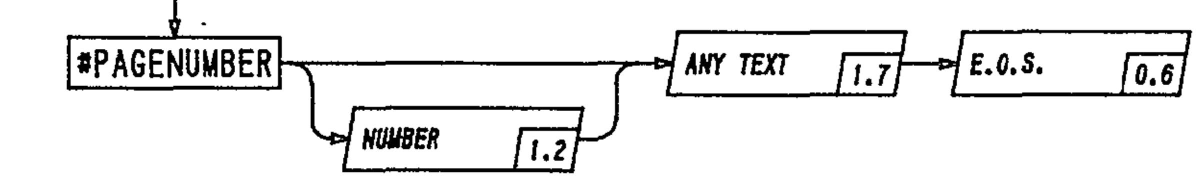
Where cards are used as the input medium, it may be desirable to have card sequence numbers at the right-hand edge of the card, in which case the #SEQUENCE directive must be used to differentiate between the input text and the sequence numbers. As shown in the syntax diagram above, the #SEQUENCE keyword may be followed by an optional integer. This integer may be used to specify the number of rightmost columns to be designated to contain sequence numbers. If no integer is supplied or a value greater than 8 is specified, the default value of eight characters, columns 73 through 80, is assumed. An integer value of zero has the effect of disabling the card sequence capability. When the #SEQUENCE capability is used, the input line (except for the sequence numbers) is handled in the usual way, and the sequence numbers are printed in the rightmost columns of the output page as determined by the #WIDTH directive (default = 80 columns). Where an input line is continued over more than one card, only the sequence number of the last card is printed.

Example: Sequence columns specification

```
#SEQUENCE 4
```

Columns 1 through 76 of the input deck are assumed to contain SDDL statements or directives, and columns 77 through 80 are assumed to contain sequence numbers.

3.13 PAGENUMBER DIRECTIVE



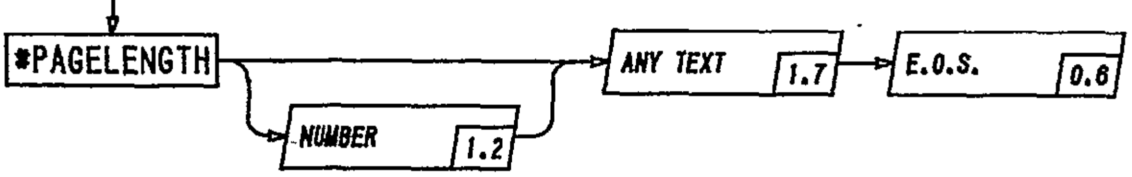
The #PAGENUMBER directive allows the user to specify the starting number which will be used for the page numbering sequence. Each time the directive is used it will cause the next page number to be set to the integer specified in the directive. Any value between 0 and 9900 is permitted. If no value is specified the default "Page 1" is assumed. The page numbering sequence may be reset as often as desired. Although duplicate sequences are permitted they should be avoided because they are confusing and detract from the document readability. This directive can be useful for segmenting the SDD.

Examples: Pagenumber directive

```
#PAGENUMBER              100
    (SDDL input of less than 100 pages)
#PAGENUMBER              200
    (SDDL input of less than 100 additional pages)
#PAGENUMBER              300
```

The purpose of this directive is to jump pagenumbers for sections of the document (e.g., 1,2,3..., 101,102,103,..., 201, 202...) or for producing documents to be inserted into other documents.

3.14 PAGELENGTH DIRECTIVE



The #PAGELENGTH directive allows the user to specify the maximum number of lines to be allowed on each page. If M is the largest number of lines that will fit on a page of output, then:

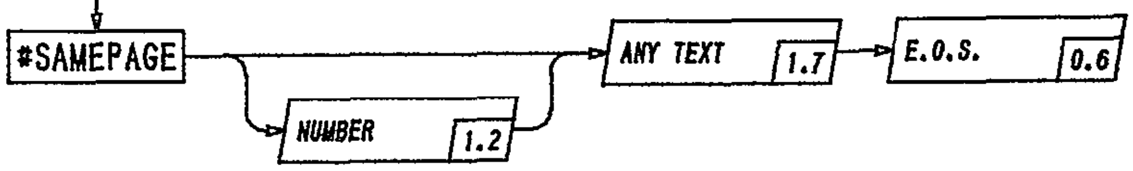
The normal or default page length = M
The allowable range for resetting the page length is 35 through M
If no integer is specified in the directive then M is assumed

Examples: Pagelength directive

```
#PAGELENGTH
#PAGELENGTH        50
```

The page length may be set and reset in this manner as often as desired without affecting other SDDL operations.

3.15 SAMEPAGE DIRECTIVE

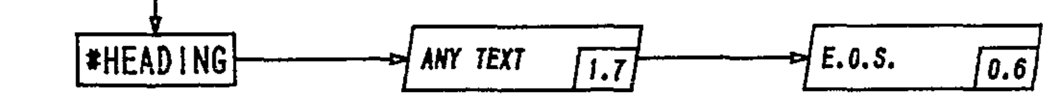


This directive can be used to reduce the size of the Software Design Document (SDD) by causing more than one module to appear on an output page. When this directive is encountered, the processor will suppress the start of a new page for as many modules as indicated by the specified integer (if no integer is specified the default = 1). Use of this directive has no effect on page overflow, page reference numbering, or the #EJECT directive. Page ejects within modules are not included in the count. As an alternative to counting modules to ascertain the correct value of n to specify, the user can bracket a group of modules by specifying a large value of n, say 1000, to turn compression "on," and a zero value of n to turn compression "off."

Examples: SAMEPAGE directive

```
#SAMEPAGE
#SAMEPAGE 2
```

3.16 HEADING DIRECTIVE



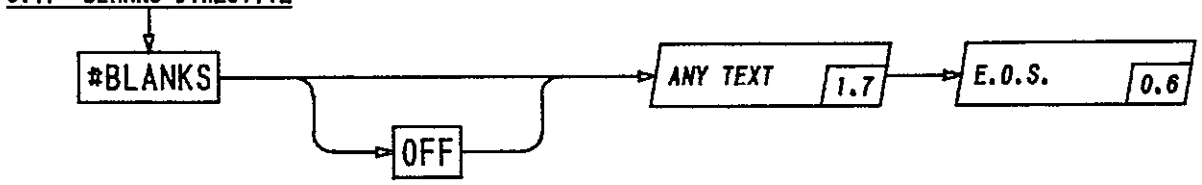
The #HEADING directive allows the user to specify a text string which the processor will then insert between the words "LINE" and "PAGE" which appear at the top of each page of the body of the SDD.

The text string which begins immediately following #HEADING and ends with the last non-blank character of the statement is centered in the heading at the top of the page. If there is insufficient space the text string is truncated on the right.

Examples: Heading directive

```
#HEADING  TEST RUN 5/1/79
#HEADING
```

3.17 BLANKS DIRECTIVE



The #BLANKS directive allows the user to specify whether the blanks preceding the first non-blank character in the input line shall be included or excluded when establishing the indentation level of the output line.

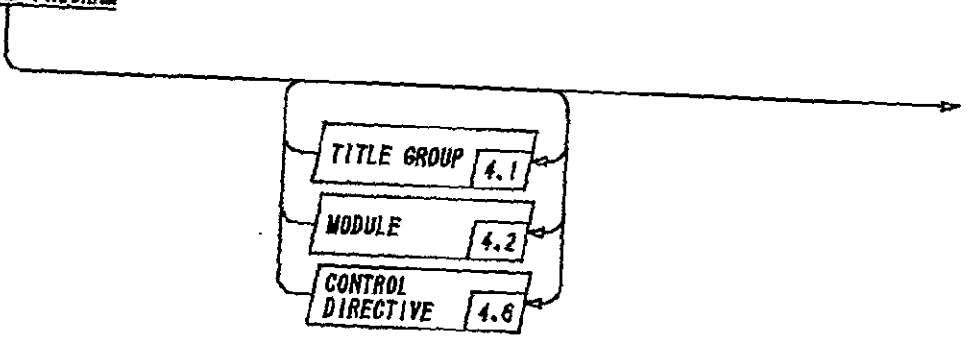
Syntax:

default mode #BLANKS #BLANKS ON #BLANKS any text	Causes preceding blanks to be left on as part of the line when establishing the indentation of the output line.
#BLANKS OFF	Causes preceding blanks to be stripped off before establishing the indentation of the output line.

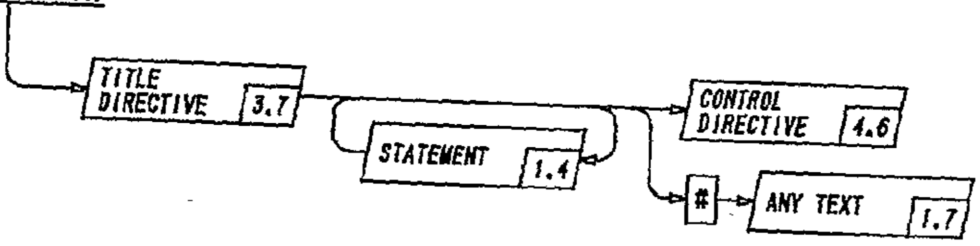
This directive may be used as often as desired to alternate between including and excluding blanks in the SDD.

4. SDDL Syntax Overview Diagrams (Level 4)

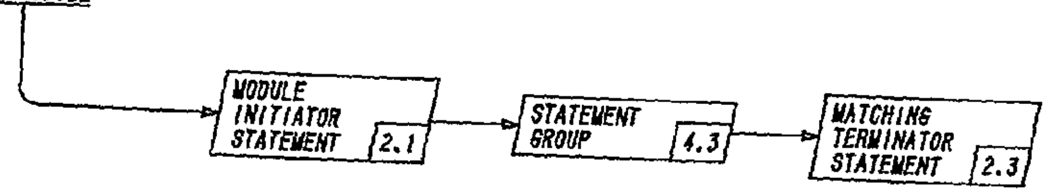
4.0 SDDL PROGRAM



4.1 TITLE GROUP

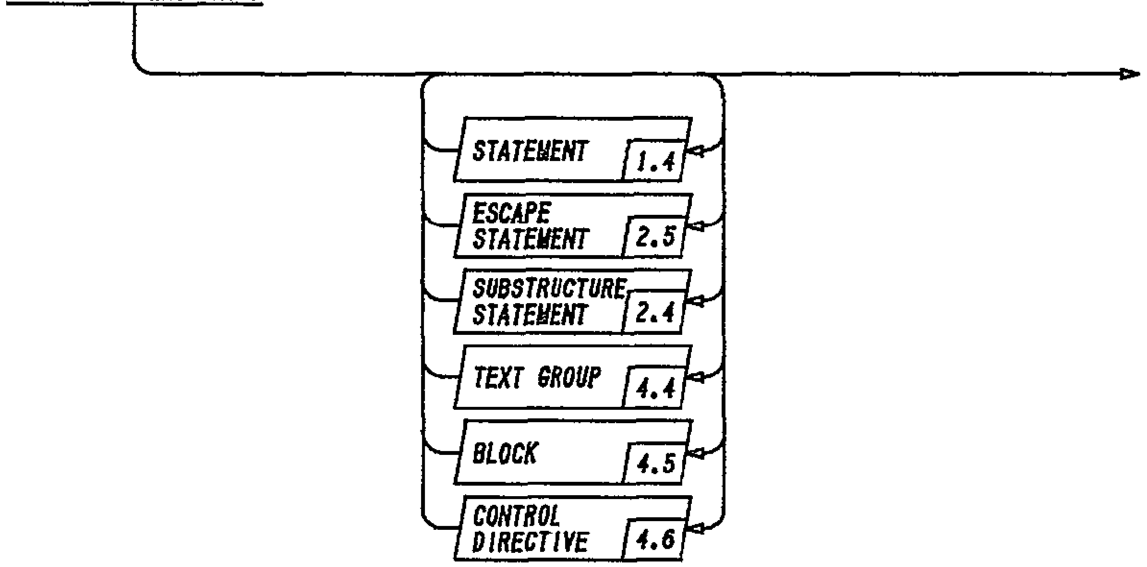


4.2 MODULE

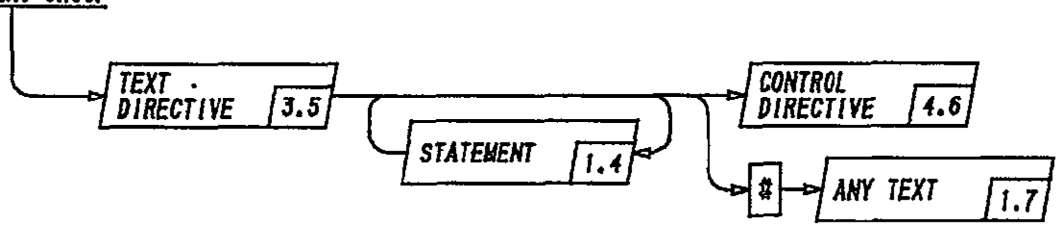


MODULE INITIATOR STATEMENT 2.1
STATEMENT GROUP 4.3
MATCHING TERMINATOR STATEMENT 2.3

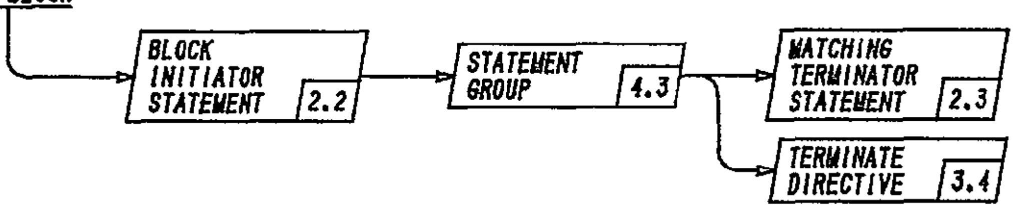
4.3 STATEMENT GROUP



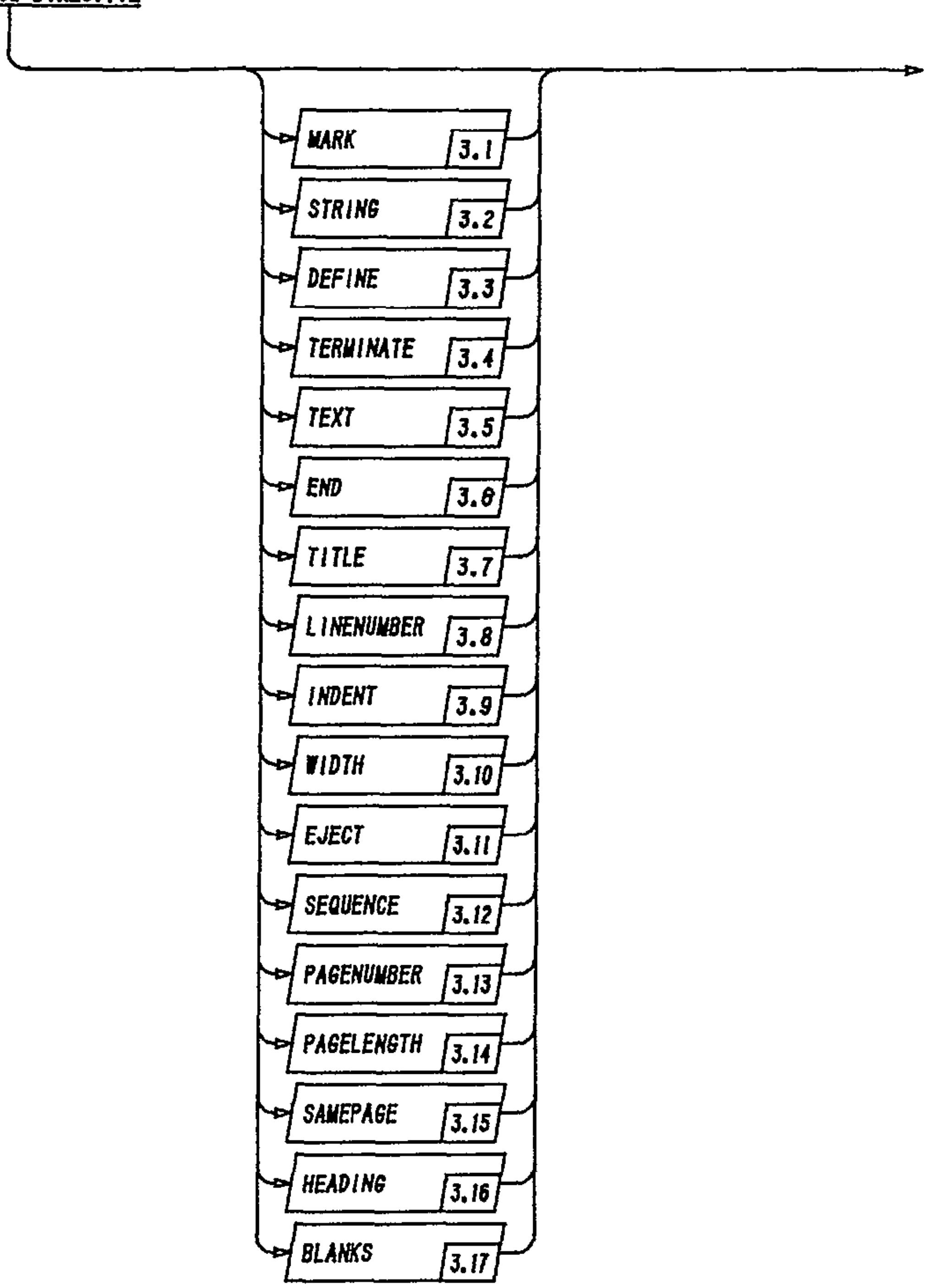
4.4 TEXT GROUP



4.5 BLOCK


BLOCK INITIATOR STATEMENT 2.2
STATEMENT GROUP 4.3
MATCHING TERMINATOR STATEMENT 2.3
TERMINATE DIRECTIVE 3.4

4.6 CONTROL DIRECTIVE



SECTION V

SAMPLE DESIGN

The following example is presented to illustrate the capability and potential of the SDDL processor. The design of the SDDL processor itself is the subject of this example. Only a small subset of the actual SDDL design is shown in order to reduce the example size to expedient proportions. Even this small, top-level portion of the SDDL processor design, however, reveals information which has an important impact on the processor.

Example: Top-level SDD for the SDDL processor:

As input:

```
#MARK REVISIONS % PROGRAM PORTABILITY CONSIDERATIONS ?
#MARK ROUTINES AND FUNCTIONS _ DATA ITEMS
#STRING DATA ITEMS "
#DEFINE BLOCK 2 LIST
#DEFINE BLOCK 2 MEMBER
#DEFINE BLOCK LOOP, , , BEGIN
#TITLE SDDL EXAMPLE

SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE


(AN ILLUSTRATION OF THE APPLICATION OF SDDL USING THE)
(SDDL PROCESSOR ITSELF AS THE OBJECT OF THE EXAMPLE. )

#END
PROGRAM OBJECTIVES
#TEXT
   THE OBJECTIVE OF SDDL IS TO PROVIDE AN EFFECTIVE COMMUNICATIONS
MEDIUM TO SUPPORT THE DESIGN AND DOCUMENTATION OF COMPLEX SOFTWARE
APPLICATIONS.  THIS OBJECTIVE IS MET BY PROVIDING:

   (1) A DESIGN AND DOCUMENTATION LANGUAGE WITH FORMS AND SYNTAX
       THAT ARE SIMPLE, UNRESTRICTIVE, AND COMMUNICATIVE

   (2) A PROCESSOR WHICH CAN CONVERT DESIGN SPECIFICATIONS INTO AN
       INTELLIGIBLE, INFORMATIVE,  MACHINE REPRODUCIBLE DOCUMENT

   (3) METHODOLOGY FOR EFFECTIVE USE OF THE LANGUAGE AND PROCESSOR

#END
PROGRAM DATA_STRUCTURE AND GLOSSARY

INPUT.TEXT.BUFFER                 A GLOBAL CHARACTER ARRAY CONTAINING
                                  A SINGLE INPUT STATEMENT FORMED BY
                                  CONCATENATION OF CONTINUED INPUT LINES

TEXT.LENGTH                       THE LENGTH OF THE CURRENT INPUT LINE
                                  (TRAILING BLANKS NOT INCLUDED)

LIST: TOKEN.DICTIONARY            LINKED LIST OF DICTIONARY ENTRIES
MEMBER ENTITY: ENTRY              POINTER TO A SINGLE DICTIONARY ENTRY
CHARACTER.COUNT                   NUMBER OF CHARACTERS IN THE ENTRY
TEXT.POINTER                      POINTER TO THE CHARACTER ARRAY
                                  CONTAINING THE TEXT OF THE ENTRY
PROGRAM.NAME                      IF ENTRY IS A KEYWORD THIS IS THE
                                  LOCATION OR IDENTIFICATION OF THE
                                  ROUTINE FOR PROCESSING THE STMT
                                  VALUE=0 IF ENTRY IS NOT A KEYWORD
```

```
LIST: REFERENCE.LIST              FIRST-IN,FIRST-OUT LIST OF
                                  REFERENCES TO THE ENTRY
MEMBER ENTITY: "REFERENCE"
PAGE.NUMBER
LINE.NUMBER
#TERMINATE 4

LIST: MODULE.STACK                PUSH DOWN STACK OF NODES REPRESENTING
                                  THE NESTED STRUCTURES OF THE DESIGN
MEMBER ENTITY: NODE
NODE.NAME                         ( IF,LOOP,PROGRAM,ETC)
INDENTATION.COLUMN
#TERMINATE 2
ENDPROGRAM DATA_STRUCTURE
PROGRAM MAIN ROUTINE
CALL INITIALIZATION ROUTINE
LOOP UNTIL ALL INPUT DATA HAS BEEN PROCESSED
CALL GET_STATEMENT # %1
*YIELD TEXT.LENGTH

CALL TOKEN_FINDER (FINDS THE FIRST TOKEN IN THE STATEMENT)
*YIELD TOKEN.TYPE

IF TOKEN.TYPE IS "IDENTIFIER"
CALL ENTABLE TO FIND THE TOKEN IN THE TOKEN.DICTIONARY
ENDIF

IF THE TOKEN WAS FOUND AND IT IS A KEYWORD
CALL KEYWORD_PROCESSOR
ELSE THE STATEMENT DOES NOT BEGIN WITH A KEYWORD
IF THE MODULE.STACK IS EMPTY
PUSH A PROGRAM MODULE ON THE MODULE.STACK
ENDIF
CALL SOURCE_LISTER TO SEND THE STATEMENT TO THE OUTPUT FILE
ENDIF

FLUSH ANY "ERROR MESSAGES" TRIGGERED BY THE STATEMENT
REPEAT
CALL WRAP_UP
EXITPROGRAM
ENDPROGRAM
PROCEDURE:  GET_STATEMENT # %1
*USING INPUT.TEXT.BUFFER
*YIELD TEXT.LENGTH

READ AN INPUT RECORD
LOOP UNTIL A NON-BLANK RECORD IS FOUND
IF THE MODULE.STACK IS NOT EMPTY (A MODULE EXISTS)
PRINT THE INPUT RECORD NUMBER AND A BLANK LINE TO THE "SDD"
ENDIF
READ ANOTHER INPUT RECORD
REPEAT
COPY THE INPUT RECORD INTO THE INPUT.TEXT.BUFFER
SET TEXT.LENGTH = "USABLE COLUMNS"( 80 - CARD SEQUENCE COLS) # ???
LOOP
FIND THE LAST NON-BLANK CHARACTER IN INPUT.TEXT.BUFFER
SET TEXT.LENGTH = COLUMN NUMBER OF THE CHARACTER
IF THE CHARACTER IS NOT A CONTINUATION.MARK
EXITPROCEDURE
ENDIF
SUBTRACT 1 FROM THE TEXT.LENGTH (BACK UP OVER THE CONTINUATION.MARK)
IF THERE IS NO MORE DATA (END OF FILE ENCOUNTERED)
EXITPROCEDURE
ENDIF
IF THE SPACE LEFT IN INPUT.TEXT.BUFFER < 80 CHARACTERS # ???
EXPAND INPUT.TEXT.BUFFER BY AT LEAST 80 CHARACTERS # ???
ENDIF
```

```
READ IN ANOTHER INPUT RECORD
COPY THE INPUT RECORD INTO INPUT.TEXT.BUFFER BEGINNING AT TEXT.LENGTH
ADD "USABLE COLUMNS" TO TEXT.LENGTH
REPEAT
ENDPROCEDURE
PROCEDURE FOR INITIALIZATION
READ IN EXECUTION TIME OPTION FLAGS FROM EXECUTION STATEMENT
   OPTION.B = BREAKPOINT
   OPTION.C = CROSS REFERENCE
   OPTION.E = "ERROR MESSAGES"
   OPTION.K = KEYWORDS
   OPTION.M = MODULE CROSS REFERENCE
   OPTION.P = PAGE REFERENCE NUMBERS
   OPTION.R = REFERENCE TREE
   OPTION.T = TABLE OF CONTENTS

IF OPTION.B IS NOT SET  BREAKPOINTING IS REQUIRED
READ IN REMAINDER OF EXECUTION STATEMENT
IF A NAME IS SPECIFIED FOR THE SDD OUTPUT FILE
SET UP A @USE RELATIONSHIP WITH SDD
ENDIF
CATALOG AND ASSIGN SDD  AS THE OUTPUT FILE
IF THE CATALOG STEP FAILED
PRINT AN ERROR MESSAGE
TERMINATE THE PROCESSOR
EXITPROCEDURE
ENDIF
BREAKPOINT THE OUTPUT TO SDD
ENDIF
ESTABLISH THE FOLLOWING MACHINE DEPENDENT CONSTANTS
  CHARACTERS.PER.WORD        = 6                          # ???
  BUFFER.COUNT               = 14 (14×6=84 CHARS/LINE)    # ???
  READ.UNIT                  = 5                          # ???
  WRITE.UNIT                 = 6                          # ???
  DEFAULT.INDENT             = 3
  RIGHT.MARGIN               = 80

INITIALIZE INPUT.TEXT.BUFFER TO AT LEAST 80 CHARACTERS   # ???
ESTABLISH TOKEN.DICTIONARY DATA STRUCTURE
CALL KEYWORD_SET_UP TO ESTABLISH DEFAULT KEYWORDS
EXITPROCEDURE
ENDPROCEDURE
PROCEDURE FOR KEYWORD_SET_UP
LOOP USING THE FOLLOWING DATA PAIRS
     ($ = POUND SIGN IN KEYWORDS BELOW)
  KEYWORD                    PROCEDURE NAME
  -------                    --------------
  $MARK                      SET_DATA_CHAR           # %1
  $STRING                    SET_STRING_CHAR         # %1
  $INDENT                    SET_INDENTATION         # %1
  $LINENUMBER                SET_LINENUMBER          # %1
  $TEXT                      BOX_TEXT                # %1
  $TITLE                     BOX_TEXT                # %1
  $END                       END_CONTROL             # %1
  $DEFINE                    DEFINE_WORDS            # %1
  $EJECT                     EJECT_PAGE              # %1
  $WIDTH                     SET_PAGE_WIDTH          # %1
  $SEQUENCE                  CARD_SEQUENCING         # %1
```

```
174        $TERMINATE                     BLIND_TERMINATOR        # %1
175
176       BEGIN ITERATION
177       FORCE THE KEYWORD INTO THE TOKEN.DICTIONARY
178       STORE THE PROCEDURE NAME INTO PROGRAM.NAME OF THE ENTRY
179       ENDLOOP
180       ENDPROCEDURE
```

As output:

```
************************************************************
*                                                          *
*        SOFTWARE DESIGN AND DOCUMENTATION LANGUAGE        *
*                                                          *
*                                                          *
*  (AN ILLUSTRATION OF THE APPLICATION OF SDDL USING THE)  *
*  (SDDL PROCESSOR ITSELF AS THE OBJECT OF THE EXAMPLE. )  *
*                                                          *
************************************************************



                              TABLE OF CONTENTS                          PAGE    I
 PAGE   LINE +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NUMBER NUMBER     MODULE NAME

   0      7  TITLE SDDL EXAMPLE
   1     16    PROGRAM OBJECTIVES
   2     31    PROGRAM DATA_STRUCTURE AND GLOSSARY
   3     63    PROGRAM MAIN ROUTINE
   4     90    PROCEDURE:  GET_STATEMENT                                        %1
   5    121    PROCEDURE FOR INITIALIZATION
   6    158    PROCEDURE FOR KEYWORD_SET_UP

   7         MODULE REFERENCE TREE

   8         MODULE - CROSS REFERENCE LISTING

   9         DATA ITEMS - CROSS REFERENCE LISTING

  11         REVISIONS - CROSS REFERENCE LISTING

  12         PROGRAM PORTABILITY CONSIDERATIONS - CROSS REFERENCE LISTING

  13         ROUTINES AND FUNCTIONS - CROSS REFERENCE LISTING
```

```
LINE                                                                          PAGE    1
  16 PROGRAM OBJECTIVES
  17    ***************************************************************************
  18    *    THE OBJECTIVE OF SDDL IS TO PROVIDE AN EFFECTIVE COMMUNICATIONS *
  19    * MEDIUM TO SUPPORT THE DESIGN AND DOCUMENTATION OF COMPLEX SOFTWARE *
  20    * APPLICATIONS.  THIS OBJECTIVE IS MET BY PROVIDING:                 *
  21    *                                                                    *
  22    *    (1) A DESIGN AND DOCUMENTATION LANGUAGE WITH FORMS AND SYNTAX   *
  23    *        THAT ARE SIMPLE, UNRESTRICTIVE, AND COMMUNICATIVE           *
  24    *                                                                    *
  25    *    (2) A PROCESSOR WHICH CAN CONVERT DESIGN SPECIFICATIONS INTO AN *
  26    *        INTELLIGIBLE, INFORMATIVE,  MACHINE REPRODUCIBLE DOCUMENT   *
  27    *                                                                    *
  28    *    (3) METHODOLOGY FOR EFFECTIVE USE OF THE LANGUAGE AND PROCESSOR *
  29    *                                                                    *
  30    ***************************************************************************
     ENDPROGRAM - STMT SUPPLIED BY PROCESSOR
```

```
LINE                                                                          PAGE    2
  31 PROGRAM DATA_STRUCTURE AND GLOSSARY
  32
  33    INPUT.TEXT.BUFFER                 A GLOBAL CHARACTER ARRAY CONTAINING
  34                                      A SINGLE INPUT STATEMENT FORMED BY
  35                                      CONCATENATION OF CONTINUED INPUT LINES
  36
  37    TEXT.LENGTH                       THE LENGTH OF THE CURRENT INPUT LINE
  38                                      (TRAILING BLANKS NOT INCLUDED)
  39
  40    LIST: TOKEN.DICTIONARY            LINKED LIST OF DICTIONARY ENTRIES
  41      MEMBER ENTITY: ENTRY              POINTER TO A SINGLE DICTIONARY ENTRY
  42        CHARACTER.COUNT                   NUMBER OF CHARACTERS IN THE ENTRY
  43        TEXT.POINTER                      POINTER TO THE CHARACTER ARRAY
  44                                          CONTAINING THE TEXT OF THE ENTRY
  45        PROGRAM.NAME                      IF ENTRY IS A KEYWORD THIS IS THE
  46                                          LOCATION OR IDENTIFICATION OF THE
  47                                          ROUTINE FOR PROCESSING THE STMT
  48                                          VALUE=0 IF ENTRY IS NOT A KEYWORD
  49        LIST: REFERENCE.LIST              FIRST-IN,FIRST-OUT LIST OF
  50                                            REFERENCES TO THE ENTRY
  51          MEMBER ENTITY: "REFERENCE"
  52            PAGE.NUMBER
  53            LINE.NUMBER
  55
  56    LIST: MODULE.STACK                PUSH DOWN STACK OF NODES REPRESENTING
  57                                        THE NESTED STRUCTURES OF THE DESIGN
  58      MEMBER ENTITY: NODE
  59        NODE.NAME                         ( IF,LOOP,PROGRAM,ETC)
  60        INDENTATION.COLUMN
  62 ENDPROGRAM DATA_STRUCTURE
```

```
LINE                                                                                PAGE    3
  63 PROGRAM MAIN ROUTINE
  64    CALL INITIALIZATION ROUTINE------------------------------------------->(  5)
  65    LOOP UNTIL ALL INPUT DATA HAS BEEN PROCESSED
  66       CALL GET_STATEMENT --------------------------------------- %1>(  4)
  67       *YIELD TEXT.LENGTH
  68
  69       CALL TOKEN_FINDER (FINDS THE FIRST TOKEN IN THE STATEMENT)----->(   )
  70       *YIELD TOKEN.TYPE
  71
  72       IF TOKEN.TYPE IS "IDENTIFIER"
  73          CALL ENTABLE TO FIND THE TOKEN IN THE TOKEN.DICTIONARY------>(   )
  74       ENDIF
  75
  76       IF THE TOKEN WAS FOUND AND IT IS A KEYWORD
  77          CALL KEYWORD_PROCESSOR--------------------------------------->(   )
  78       ELSE THE STATEMENT DOES NOT BEGIN WITH A KEYWORD
  79          IF THE MODULE.STACK IS EMPTY
  80             PUSH A PROGRAM MODULE ON THE MODULE.STACK
  81          ENDIF
  82          CALL SOURCE_LISTER TO SEND THE STATEMENT TO THE OUTPUT FILE->(   )
  83       ENDIF
  84
  85       FLUSH ANY "ERROR MESSAGES" TRIGGERED BY THE STATEMENT
  86    REPEAT
  87    CALL WRAP_UP---------------------------------------------------------->(   )
  88 <--EXITPROGRAM
  89 ENDPROGRAM
```

```
LINE                                                                             PAGE    4
PROCEDURE:  GET_STATEMENT                                                               %1
   *USING INPUT.TEXT.BUFFER
   *YIELD TEXT.LENGTH

   READ AN INPUT RECORD
   LOOP UNTIL A NON-BLANK RECORD IS FOUND
      IF THE MODULE.STACK IS NOT EMPTY (A MODULE EXISTS)
         PRINT THE INPUT RECORD NUMBER AND A BLANK LINE TO THE "SDD"
      ENDIF
      READ ANOTHER INPUT RECORD
   REPEAT
   COPY THE INPUT RECORD INTO THE INPUT.TEXT.BUFFER
   SET TEXT.LENGTH = "USABLE COLUMNS"( 80 - CARD SEQUENCE COLS)                    ???
   LOOP
      FIND THE LAST NON-BLANK CHARACTER IN INPUT.TEXT.BUFFER
      SET TEXT.LENGTH = COLUMN NUMBER OF THE CHARACTER
      IF THE CHARACTER IS NOT A CONTINUATION.MARK
<---------EXITPROCEDURE
      ENDIF
      SUBTRACT 1 FROM THE TEXT.LENGTH (BACK UP OVER THE CONTINUATION.MARK)
      IF THERE IS NO MORE DATA (END OF FILE ENCOUNTERED)
<---------EXITPROCEDURE
      ENDIF
      IF THE SPACE LEFT IN INPUT.TEXT.BUFFER < 80 CHARACTERS                       ???
         EXPAND INPUT.TEXT.BUFFER BY AT LEAST 80 CHARACTERS                        ???
      ENDIF
      READ IN ANOTHER INPUT RECORD
      COPY THE INPUT RECORD INTO INPUT.TEXT.BUFFER BEGINNING AT TEXT.LENGTH
      ADD "USABLE COLUMNS" TO TEXT.LENGTH
   REPEAT
ENDPROCEDURE
```

```
PROCEDURE FOR INITIALIZATION
   READ IN EXECUTION TIME OPTION FLAGS FROM EXECUTION STATEMENT
      OPTION.B = BREAKPOINT
      OPTION.C = CROSS REFERENCE
      OPTION.E = "ERROR MESSAGES"
      OPTION.K = KEYWORDS
      OPTION.M = MODULE CROSS REFERENCE
      OPTION.P = PAGE REFERENCE NUMBERS
      OPTION.R = REFERENCE TREE
      OPTION.T = TABLE OF CONTENTS

   IF OPTION.B IS NOT SET  BREAKPOINTING IS REQUIRED
      READ IN REMAINDER OF EXECUTION STATEMENT
      IF A NAME IS SPECIFIED FOR THE SDD OUTPUT FILE
         SET UP A @USE RELATIONSHIP WITH SDD
      ENDIF
      CATALOG AND ASSIGN SDD  AS THE OUTPUT FILE
      IF THE CATALOG STEP FAILED
         PRINT AN ERROR MESSAGE
         TERMINATE THE PROCESSOR
<--------EXITPROCEDURE
      ENDIF
      BREAKPOINT THE OUTPUT TO SDD
   ENDIF
   ESTABLISH THE FOLLOWING MACHINE DEPENDENT CONSTANTS
     CHARACTERS.PER.WORD        = 6                                      ???
     BUFFER.COUNT               = 14 (14*6=84 CHARS/LINE)                ???
     READ.UNIT                  = 5                                      ???
     WRITE.UNIT                 = 6                                      ???
     DEFAULT.INDENT             = 3
     RIGHT.MARGIN               = 80

   INITIALIZE INPUT.TEXT.BUFFER TO AT LEAST 80 CHARACTERS                ???
   ESTABLISH TOKEN.DICTIONARY DATA STRUCTURE
   CALL KEYWORD_SET_UP TO ESTABLISH DEFAULT KEYWORDS------------------>(  6)
<--EXITPROCEDURE
ENDPROCEDURE
```

```
PROCEDURE FOR KEYWORD_SET_UP
   LOOP USING THE FOLLOWING DATA PAIRS
           ($ = POUND SIGN IN KEYWORDS BELOW)
        KEYWORD                    PROCEDURE NAME
        -------                    --------------
        $MARK                      SET_DATA_CHAR                         %1
        $STRING                    SET_STRING_CHAR                       %1
        $INDENT                    SET_INDENTATION                       %1
        $LINENUMBER                SET_LINENUMBER                        %1
        $TEXT                      BOX_TEXT                              %1
        $TITLE                     BOX_TEXT                              %1
        $END                       END_CONTROL                           %1
        $DEFINE                    DEFINE_WORDS                          %1
        $EJECT                     EJECT_PAGE                            %1
        $WIDTH                     SET_PAGE_WIDTH                        %1
        $SEQUENCE                  CARD_SEQUENCING                       %1
        $TERMINATE                 BLIND_TERMINATOR                      %1

   BEGIN ITERATION
      FORCE THE KEYWORD INTO THE TOKEN.DICTIONARY
      STORE THE PROCEDURE NAME INTO PROGRAM.NAME OF THE ENTRY
   ENDLOOP
ENDPROCEDURE
```

```
   **************  MODULE REFERENCE TREE  ******                       PAGE    7
 LN  PAGE
  1     1  OBJECTIVES

  2     2  DATA_STRUCTURE

  3     3  MAIN
  4     5  .  INITIALIZATION
  5     6  .  .  KEYWORD_SET_UP
  6     4  .  GET_STATEMENT
  7     *  .  TOKEN_FINDER
  8     *  .  ENTABLE
  9     *  .  KEYWORD_PROCESSOR
 10     *  .  SOURCE_LISTER
 11     *  .  WRAP_UP

                                        MODULE
                                CROSS REFERENCE LISTING                    PAGE    8
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                                 LINE NUMBERS

DATA_STRUCTURE
    PAGE   2  PROGRAM DATA_STRUCTURE                         31   62
ENTABLE
    PAGE   3  PROGRAM MAIN                                   73
GET_STATEMENT
    PAGE   3  PROGRAM MAIN                                   66
    PAGE   4  PROCEDURE:  GET_STATEMENT                      90
INITIALIZATION
    PAGE   3  PROGRAM MAIN                                   64
    PAGE   5  PROCEDURE FOR INITIALIZATION                  121
KEYWORD_PROCESSOR
    PAGE   3  PROGRAM MAIN                                   77
KEYWORD_SET_UP
    PAGE   5  PROCEDURE FOR INITIALIZATION                  155
    PAGE   6  PROCEDURE FOR KEYWORD_SET_UP                  158
MAIN
    PAGE   3  PROGRAM MAIN                                   63
OBJECTIVES
    PAGE   1  PROGRAM OBJECTIVES                             16
SOURCE_LISTER
    PAGE   3  PROGRAM MAIN                                   82
TOKEN_FINDER
    PAGE   3  PROGRAM MAIN                                   69
WRAP_UP
    PAGE   3  PROGRAM MAIN                                   87
```

```
                                     DATA ITEMS
                               CROSS REFERENCE LISTING                    PAGE    9
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                              LINE NUMBERS

BUFFER.COUNT
    PAGE   5  PROCEDURE FOR INITIALIZATION               147
CHARACTERS.PER.WORD
    PAGE   5  PROCEDURE FOR INITIALIZATION               146
CHARACTER.COUNT
    PAGE   2  PROGRAM DATA_STRUCTURE                      42
CONTINUATION.MARK
    PAGE   4  PROCEDURE:  GET_STATEMENT                  106  109
DEFAULT.INDENT
    PAGE   5  PROCEDURE FOR INITIALIZATION               150
ERROR MESSAGES
    PAGE   3  PROGRAM MAIN                                85
    PAGE   5  PROCEDURE FOR INITIALIZATION               125
IDENTIFIER
    PAGE   3  PROGRAM MAIN                                72
INDENTATION.COLUMN
    PAGE   2  PROGRAM DATA_STRUCTURE                      60
INPUT.TEXT.BUFFER
    PAGE   2  PROGRAM DATA_STRUCTURE                      33
    PAGE   4  PROCEDURE:  GET_STATEMENT                   91  101  104  113  114  117
    PAGE   5  PROCEDURE FOR INITIALIZATION               153
LINE.NUMBER
    PAGE   2  PROGRAM DATA_STRUCTURE                      53
MODULE.STACK
    PAGE   2  PROGRAM DATA_STRUCTURE                      56
    PAGE   3  PROGRAM MAIN                                79   80
    PAGE   4  PROCEDURE:  GET_STATEMENT                   96
NODE.NAME
    PAGE   2  PROGRAM DATA_STRUCTURE                      59
OPTION.B
    PAGE   5  PROCEDURE FOR INITIALIZATION               123  132
OPTION.C
    PAGE   5  PROCEDURE FOR INITIALIZATION               124
OPTION.E
    PAGE   5  PROCEDURE FOR INITIALIZATION               125
OPTION.K
    PAGE   5  PROCEDURE FOR INITIALIZATION               126
OPTION.M
    PAGE   5  PROCEDURE FOR INITIALIZATION               127
OPTION.P
    PAGE   5  PROCEDURE FOR INITIALIZATION               128
OPTION.R
    PAGE   5  PROCEDURE FOR INITIALIZATION               129
OPTION.T
    PAGE   5  PROCEDURE FOR INITIALIZATION               130
PAGE.NUMBER
    PAGE   2  PROGRAM DATA_STRUCTURE                      52
PROGRAM.NAME
    PAGE   2  PROGRAM DATA_STRUCTURE                      45
    PAGE   6  PROCEDURE FOR KEYWORD_SET_UP               178
READ.UNIT
    PAGE   5  PROCEDURE FOR INITIALIZATION               148
```

```
                                 DATA ITEMS
                           CROSS REFERENCE LISTING                   PAGE  10
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                           LINE NUMBERS

REFERENCE
    PAGE  2  PROGRAM DATA_STRUCTURE                   51
    PAGE  5  PROCEDURE FOR INITIALIZATION            124  127  128  129
REFERENCE.LIST
    PAGE  2  PROGRAM DATA_STRUCTURE                   49
RIGHT.MARGIN
    PAGE  5  PROCEDURE FOR INITIALIZATION            151
SDD
    PAGE  4  PROCEDURE:  GET_STATEMENT                97
    PAGE  5  PROCEDURE FOR INITIALIZATION            134  135  137  143
TEXT.LENGTH
    PAGE  2  PROGRAM DATA_STRUCTURE                   37
    PAGE  3  PROGRAM MAIN                             67
    PAGE  4  PROCEDURE:  GET_STATEMENT                92  102  105  109  117  118
TEXT.POINTER
    PAGE  2  PROGRAM DATA_STRUCTURE                   43
TOKEN.DICTIONARY
    PAGE  2  PROGRAM DATA_STRUCTURE                   40
    PAGE  3  PROGRAM MAIN                             73
    PAGE  5  PROCEDURE FOR INITIALIZATION            154
    PAGE  6  PROCEDURE FOR KEYWORD_SET_UP            177
TOKEN.TYPE
    PAGE  3  PROGRAM MAIN                             70   72
USABLE COLUMNS
    PAGE  4  PROCEDURE:  GET_STATEMENT               102  118
WRITE.UNIT
    PAGE  5  PROCEDURE FOR INITIALIZATION            149
```

```
                                  REVISIONS
                           CROSS REFERENCE LISTING                   PAGE  11
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                           LINE NUMBERS

%1
    PAGE  3  PROGRAM MAIN                             66
    PAGE  4  PROCEDURE:  GET_STATEMENT                90
    PAGE  6  PROCEDURE FOR KEYWORD_SET_UP            163  164  165  166  167  168
               169  170  171  172  173  174
```

```
                         PROGRAM PORTABILITY CONSIDERATIONS
                              CROSS REFERENCE LISTING                PAGE   12
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                               LINE NUMBERS

???
    PAGE    4  PROCEDURE:  GET_STATEMENT                  102  113  114
    PAGE    5  PROCEDURE FOR INITIALIZATION               146  147  148  149  153



                              ROUTINES AND FUNCTIONS
                              CROSS REFERENCE LISTING                PAGE   13
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
IDENTIFIER    MODULE NAME                               LINE NUMBERS

BLIND_TERMINATOR
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               174
BOX_TEXT
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               167  168
CARD_SEQUENCING
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               173
DEFINE_WORDS
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               170
EJECT_PAGE
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               171
END_CONTROL
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               169
SET_DATA_CHAR
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               163
SET_INDENTATION
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               165
SET_LINENUMBER
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               166
SET_PAGE_WIDTH
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               172
SET_STRING_CHAR
    PAGE    6  PROCEDURE FOR KEYWORD_SET_UP               164
```

SECTION VI

USING THE SDDL PROCESSOR

A. RUN-TIME PROCESSOR CONTROL OPTIONS

Run-time control options permit the user to cause certain processor functions, listed below, to be suppressed or altered. These options are invoked by adding the appropriate letter keys to the SDDL execution statement and remain in effect throughout the execution of the program. The letter keys, shown below in Table 6-1, may be given in any order.

Table 6-1. SDDL Run Time Option Summary

Option Letter Key	Meaning
B	Breakpoint operation (available only on the UNIVAC implementation) is suppressed
C	Cross reference tables for all marks and strings are suppressed
D	Do-nothing (i.e., passive, non-keyword) statements are omitted from the body of the SDD
E	Error messages are suppressed
K	Keyword definition (for the default set, see Table 2-1) is suppressed
M	Module cross reference table is suppressed
N	Null-titled, cross reference table output is suppressed
P	Page reference numbers on module invocation statements are omitted
R	Reference tree of forward calls to modules is suppressed
T	Table of contents is omitted
F	FORTRAN option. The processor is configured to handle input of FORTRAN programs

Characters other than a letter key corresponding to one of the available options will be ignored. Note that, with only one exception (F), the option meanings are consistent in that they all cause the suppression or omission of a processor function. Thus if no options are specified, the processor will perform all of its functions.

1. B Option - BREAKPOINT Suppression

This option only applies to the UNIVAC 1108 implementation, which requires that the SDDL output be breakpointed to a print file. Normally, the processor performs all of the steps necessary to do the breakpoint operation, but occasionally, for a quick look at a small part of the output, it is convenient to have the output come directly to an interactive terminal. Use of the B option for this purpose will cause the input and output streams to be merged together on the terminal screen/paper. Since the processor always reads ahead one statement, the user will be required to enter input one statement ahead of the processing.

2. C Option - Cross Reference Tables Suppression

This option will suppress the output of all the cross reference tables.

3. D Option - Do-Nothing Statement Suppression

This option will cause the processor to suppress the output of all lines which do not begin with an SDDL keyword. The D option may be used to reduce the volume of the output in situations where the user is only interested in seeing the program's flow of control. This option directs the processor to output only those lines which begin with keywords such as IF, ELSE, ENDIF, LOOP, CYCLE, ENDLOOP, PROGRAM, RETURN, ENDPROGRAM, and most importantly CALL. These statements show the logical flow of the program control, including page reference numbers for subroutine calls. Since the size of the modules may be considerably reduced, the user may wish to use the #SAMEPAGE directive to suppress page ejects between modules.

4. E Option - Error Message Suppression

The error messages pertaining to nested structures closed automatically, keywords used out of context, and syntax errors on SDDL directive specifications are omitted from the output. Incorrectly specified SDDL directives are always listed in the output, but with the E option in effect the accompanying error message will be omitted.

5. K Option - Keyword Suppression

This option, which causes the processor to bypass definition of default keywords (see Table 2-1), can be useful in situations where most or all of the default keywords are inappropriate. Use of this option will obviate the need to explicitly null out all the default keywords with the #DEFINE NULL directive.

6. M Option - <u>M</u>odule Cross Reference Table Suppression

The module cross reference table is a list of all the modules encountered, either defined or called, in your program. The modules are listed alphabetically with every occurrence referenced by module name, page number, and line number.

7. N Option - <u>N</u>ull Titled Cross Reference Table Is Suppressed

This option causes the processor to omit the output of the cross reference table for which a null or blank title has been specified. The text encountered between the directive keyword, #MARK, and the punctuation symbols being specified is recognized as the title of the cross reference table. If this space is left blank (no title specified, blank assumed), then the execution time N option will suppress the printout of the cross reference table for those marks.

Example: Cross reference table with a null title

```
#MARK    % ?
```

Note that both the % and the ? have the same title (i.e., blank or null), and therefore the N option will suppress the cross reference table associated with these marks.

8. P Option - <u>P</u>age Reference Numbers on Module Invocation Statements

When the processor encounters a module invocation statement it prints a right arrow extending from the last non-blank character in the line to a pair of parentheses at the right-hand margin of the document. In the parentheses the processor places the page number where the invoked module is defined. If the module was defined prior to its invocation, its page number is known and is printed in place with the rest of the statement. If the module has not been defined, however, its page number cannot be known and placement of the reference number must be deferred. Thus, the processor must make a second (automatic) pass over the output file to supply the missing page reference numbers. If page reference numbers are not needed, as in a test run, the user may suppress the second pass operation with the P option.

Note that the information contained in the table of contents also cannot be known until all of the input has been processed, and therefore it must be the last output written to the SDD. This means that the second pass operation must also move the table of contents to the front of the output file. Thus if the P option is used to suppress page reference numbers, it will also have the side effect of printing the table of contents at the end of the output.

9. R Option - Reference Tree of Forward Calls to Modules

This table displays the module invocation hierarchy in a tree format. Each module named in the document appears in this table in relation to where it was invoked in the overall structure. The relationship between the modules is shown by listing the called modules below and indented one level to the right of the module in which it was called. This results in a cascade of indentation (modules may appear more than once in the table) which displays the calling hierarchy of the document. The R option suppresses this table.

10. T Option - Table of Contents

The T option suppresses the output of the table of contents. (Note that use of the P option will cause the table of contents to be printed at the end instead of the beginning of the document.)

11. F Option - FORTRAN Option

The F option may be used for processing FORTRAN programs with SDDL in order to obtain a table of contents, cross reference tables, module reference tree, and the module cross reference table. When exercised, this option directs the processor to exclude columns 1 through 6 of the input line when interpreting the meaning of the statement. Thus the input line is considered to begin in column 7, which is the FORTRAN convention.

The user must bear in mind that this convention will also apply to SDDL directives, which therefore must begin at column 7 or beyond.

The SDDL processor copies columns 1-6 of the input deck onto the output listing, placing them at the left, between the input line numbers and the left margin of the body of the text.

The F option also establishes the following default SDDL keywords in place of the ones listed in Table 2-1.

Type	:	Initiator	Terminator	Escape	Substructure
Module	:	SUBROUTINE	END	RETURN	ENTRY
Module	:	FUNCTION	ENDFUNCTION		
Module	:	PROCEDURE	ENDPROCEDURE	EXITPROCEDURE	
Block	:	DO	CONTINUE		
Block	:	IF	ENDIF	ELSE	
Call	:	CALL, GO			

B. UNIVAC IMPLEMENTATION EXECUTION PROCEDURE

After the SDDL input has been loaded into one or more elements (say QUAL*FILE.IN1, QUAL*FILE.IN2), it is processed and printed by entering the following EXEC 8 commands:

```
@SDDL,[options] [SDD-output-file name.]
@ADD QUAL*FILE.IN1
@ADD QUAL*FILE.IN2
@FREE SDD$.
@SYM SDD$.......
```

The output file name specification is optional (default = SDD$.) but if supplied it must have a terminal period, e.g., "SDDTEST.," to indicate that it is a file. It need not be cataloged or assigned prior to use since these functions, and the breakpointing, will be performed by the processor. The processor will execute an EXEC 8 @USE command to relate the user's output file name to SDD$.

Example: UNIVAC execution procedure with options

```
@SDDL,CMR    TEMP.
@ADD QUAL*FN.A
@ADD QUAL*FN.B
@FREE TEMP.
@SYM,U TEMP.,HOLD/HOLD,G9HSPA
```

In the above example the C, M, and R run time options are exercised and TEMP. is established as the output file name. Two input elements are fed into the processor with the EXEC 8 @ADD command and the output file is freed and printed.

Example: UNIVAC execution procedure, no options, no file name

```
@SDDL
@ADD Q*F.A
#LINENUMBER
@ADD Q*F.B
@FREE SDD$.
@SYM SDD$., HOLD/HOLD, G9HSPA
```

This example exercises no run-time options and defaults the output file name to SDD$. It also utilizes the #LINENUMBER directive (which could equally well have been internal to the input element) between the input elements to restart the line numbering sequence.

C. JCL REQUIRED FOR THE EXECUTION OF SDDL IN AN IBM OS ENVIRONMENT

```
//jobname job
```

Job card

```
//stepname EXEC PGM=SDDL,REGION=150K,PARM=options
```

In some facilities the region parameter may be omitted

```
//STEPLIB DD program object library
```

The STEPLIB DD card permits the SDDL processor to be loaded from a data set other than SYS1.LINKLIB. If your copy of SDDL has been loaded onto SYS1.LINKLIB, then the STEPLIB DD card may be omitted.

```
//SIMU05 DD DUMMY
```

This data set is not used when SDDL is executed in batch mode, but because an open is issued, the DD card is required.

```
//SIMU11 DD DCB=(RECFM=FBA,LRECL=133,BLKSIZE=nnn),
            SPACE=(TRK,(50,50)), UNIT=SYSDA
```

This DD card specifies an intermediate scratch file that is written using this DDNAME and read back using SIMU12. Since the DSN parameter has been omitted, a temporary data set is created with the DISP of (NEW,DELETE,DELETE). If it is desired that this data set be obtained from a specific device, then the UNIT=SYSDA must be replaced by parameters specifying the device type and identifying the VOLUME. The BLKSIZE should be chosen as an integer multiple of the logical record length (133) in order to make efficient use of the disk space. The maximum BLKSIZE varies with the disk system.

```
//SIMU12 DD DCB=(RECFM=FB,LRECL=133,BLKSIZE=nnn),
            DSN=*.SIMU11,VOL=REF=*.SIMU11,DISP=OLD
```

The scratch file written out using SIMU11 is read back through this DDNAME. The DCB parameters must be identical <u>except for the RECFM parameter, which must be different, as shown</u>. The DSN must be equated to that assigned by the system to SIMU11, and it must be allocated to the same physical volume. Since the data set was created by the previous DD statement, the disposition parameter must be coded as DISP=OLD.

```
//SIMU10 DD SYSOUT=A,DCB=(RECFM=FBA,LRECL=133,BLKSIZE=nnn)
```

The output listing is written to SIMU10. Since the record format must be RECFM=FBA, it is necessary to supply all of the DCB parameters. A convenient choice would be the DCB parameters used in SIMU11. Should the default space in your facility for a SYSOUT data set be very small, or your design very large, it may be necessary to explicitly provide SPACE parameters.

```
//SIMU06 DD SYSOUT=A
```

The SDDL normal termination message and SDDL processor malfunction messages are output to this file.

```
//SIMU09 DD*
```

SIMU09 is the SDDL input file. When an input deck is to be included with the job stream, the DD card should be coded as above. As shown below, the source may also be obtained from a partitioned data set or any serial data set containing logical records of length 80 bytes.

```
//SIMU09 DD UNIT=TAPE,VOL=SER=999999,DCB=(RECFM=FM,LRECL=80,
          BLKSIZE=800),LABEL=(,SL),DSN=INCARDS,DISP=(OLD,KEEP)
```

The input DD statement may be written as above to take the source from a magnetic tape.

```
//SIMU09 DD DSN=SDDLS(PROGRAMA),DISP=SHR
```

The above DD statement will select the member "PROGRAMA" from the cataloged partitioned data set "SDDLS."

Kernighan, B. W., and Plauger, P. J., The Elements of Programming Style, McGraw-Hill Book Co. New York, 1974, pp. 36-39.

Kleine, H., and Morris, R. V., "Modern Programming: A Definition," SIGPLAN Notices, Vol. 9, No. 9, Sept. 1974, pp. 14-17.

Kleine, H., "Automating the Software Design Process by Means of the Software Design and Documentation Language", Proceedings of the No. 15 Design Automation Conference, IEEE Catalog #78 Ch. 1363-1C, Las Vegas, Nev., June 1978, 371-379.

Kleine, H., "A Vehicle for Developing Standards for Simulation Programming", Proceedings of Winter, 77 Simulation Conference, Highland, Sargent, and Schmidt, eds., 731-741.

Liskov, B., and Zilles, S., "Programming with Abstract Data Types," SIGPLAN Notices, March 1974, pp. 50-59.

Luppino, F. B., and Smith, R. L., "Programming Support Library Functional Requirements," Vol. V of Structured Programming Series, RADC-TR-74-300, U. S. Air Force, July 25, 1974.

Miller, E. F., Jr., A Compendium of Language Extensions to Support Structured Programming, RN-42, General Research Corp., Santa Barbara, CA, Jan. 1973.

Mills, H. D., "Top-Down Programming in Large Systems," in Debugging Techniques in Large Systems, Edited by R. Rustin, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971, pp. 43-45.

Mills, H. D., Mathematical Foundations of Structured Programming, IBM Document FSC 72-6012, IBM Federal Systems Division, Gaithersburg, MD, Feb. 1972.

Myers, G. J., Composite Design: The Design of Modular Programs, Technical Report TR00.2406, IBM, Poughkeepsie, N. Y., Jan. 29, 1973.

Ogdin, C.A., Software Design for Microcomputers, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.

Robert, D. C., "File Organization Techniques," Advances in Computers, Vol. 12, Academic Press, New York, 1972.

Shneiderman, B., "A Review of Design Techniques for Programs and Data," Software-Practice and Experience, Vol. 6, 1976, pp. 555-567.

Shneiderman, B., et al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," Communications of the ACM, Vol. 20, No. 6, June 1977, pp. 373-381.

Tausworthe, R. C., Standardized Development of Computer Software. Part 1, Methods, SP 43-29, Jet Propulsion Laboratory, Pasadena, CA, July 1976.

BIBLIOGRAPHY

Baker, F. T., "Structured Programming in a Production Programming Environment," IEEE Trans. on Software Engr., Vol. SE-1, No. 2, pp. 241-252, June 1975.

Baker, F. T., and Mills, H. D., "Chief Programmer Teams," Datamation, Vol. 19, No. 12, pp. 58-61, Dec. 1973.

Basili, V. R., SIMPL-X, A Language for Writing Structured Programs, Nat. Tech. Info. Service Report AD755-703, U.S. Dept. of Commerce, Springfield, VA, Jan. 1973.

Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973.

Brinch Hansen, P., "The Purpose of Concurrent Pascal," Proceedings of the 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CH0940-7CSR, pp. 305-309. (Also published in SIGPLAN Notices, June 1975, pp. 305-309.)

Brinch Hansen, P., Concurrent Pascal: A Programming Language for Operating System Design, California Institute of Technology Information Science Technical Report No. 10, Pasadena, CA, April 1974.

Brooks, F. P., "The Mythical Man-Month," Datamation, Vol. 20, No. 12, pp. 45-52, Dec. 1974.

Caine, S. H., and Gordon, E. K., "PDL--A Tool for Software Design," Program Design Language Reference Guide, Caine, Farber, and Gordon, Inc., Pasadena, CA, Sept. 18, 1974.

Constantine, L. L., Fundamentals of Program Design, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

Dahl, O. J., and Hoare, C. A. R., "Hierarchical Program Structures," in Structured Programming, Academic Press, New York, 1972.

Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, Academic Press, New York, 1972 (particularly pp. 16-23).

Flynn, J., SFTRAN User's Guide, Comput. Memo. 914-337, Jet Propulsion Laboratory, Pasadena, CA, July 1973 (JPL internal document).

Heimburger, D. A., et al, "VEEP: Vehicle Economy, Emissions and Performance Program" Proceedings of Winter, 77 Simulation Conference, Highland, Sargent, and Schmidt, eds.

Hoare, C. A. R., "Notes on Data Structuring," in Structured Programming, Academic Press, New York, 1972.

Katzan, H., Jr., Advanced Programming, D. Van Nostrand Reinhold Co., NJ, 1970, pp. 153-163.